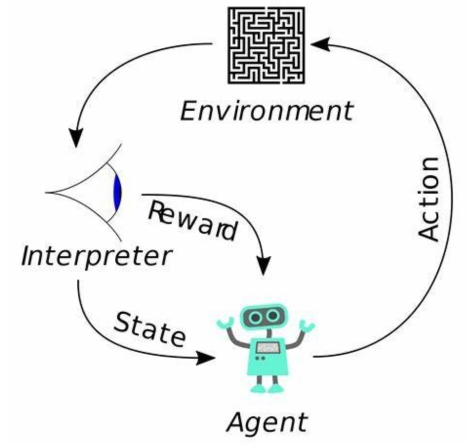
Reinforcement Learning



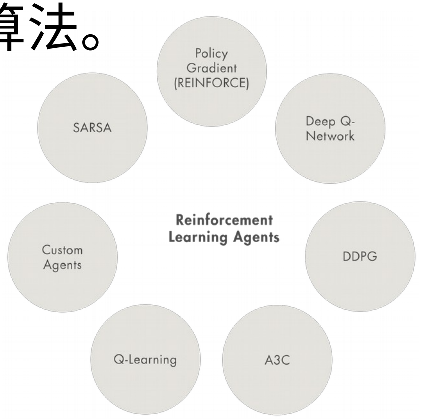
关于 RL&DRL

强化学习是机器学习的一个分支，它可以针对复杂系统（如机器人和自主系统）实现控制器和决策系统。借助深度强化学习，可以实现深度神经网络，这类网络使用从仿真系统或物理系统动态生成的数据进行训练，从而学习复杂行为。与其他机器学习方法不同，深度强化学习不需要预定义的标注或未标注的训练数据集。通常，只需要一个表示环境的仿真模型。

深度强化学习智能体

深度强化学习智能体由深度神经网络策略和算法构成，其中策略用于将输入状态映射到输出动作，算法负责更新此策略。常见算法包括深度 Q 网络 (DQN)、深度确定性策略梯度 (DDPG)、软执行器评价器 (SAC) 和近端策略优化 (PPO)。算法会基于从环境中采集的观测值和奖励来更新策略，以最大化预期的长期奖励。

Reinforcement Learning Toolbox 可以用编程方式或交互方式（使用强化学习设计器）创建深度强化学习智能体。可以从现成的热门算法中选择，也可以使用已有模板和示例实现自定义算法。



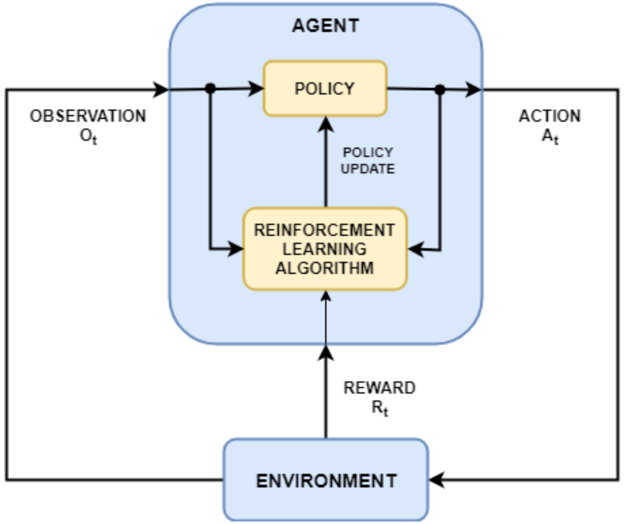
使用 MATLAB 和 Simulink 进行环境建模

深度强化学习算法训练是一个动态过程，因为智能体需要与周边环境进行交互。对于机器人和自主系统等应用形式，使用实际硬件执行此类训练不仅代价高昂，还可能面临危险。因此，人们倾向于采用通过仿真生成数据的虚拟环境模型来进行深度强化学习。

可以使用 MATLAB 和 Simulink 构建环境模型，以描述系统动态、智能体的动作对系统动态产生的影响，以及用于评估所执行动作的优度的奖励。这些模型在本质上可以是连续的或离散的，能够以不同的保真度表示系统。此外，可以通过并行仿真来加快训练。在某些情况下，可以重用现有的 MATLAB 和 Simulink 系统模型，只需稍加改动即可将其用于深度强化学习。

What Is Reinforcement Learning

Reinforcement learning is a goal-directed computational approach where a computer learns to perform a task by interacting with an unknown dynamic environment. This learning approach enables a computer to make a series of decisions to maximize the cumulative reward for the task without human intervention and without being explicitly programmed to achieve the task.



What Is Reinforcement Learning

The goal of reinforcement learning is to train an agent to complete a task within an unknown environment. The agent receives observations and a reward from the environment and sends actions to the environment. The reward is a measure of how successful an action is with respect to completing the task goal.

The agent contains two components: a policy and a learning algorithm.

- The policy is a mapping that selects actions based on the observations from the environment. Typically, the policy is a function approximator with tunable parameters, such as a deep neural network.
- The learning algorithm continuously updates the policy parameters based on the actions, observations, and reward. The goal of the learning algorithm is to find an optimal policy that maximizes the cumulative reward received during the task.

What Is Reinforcement Learning

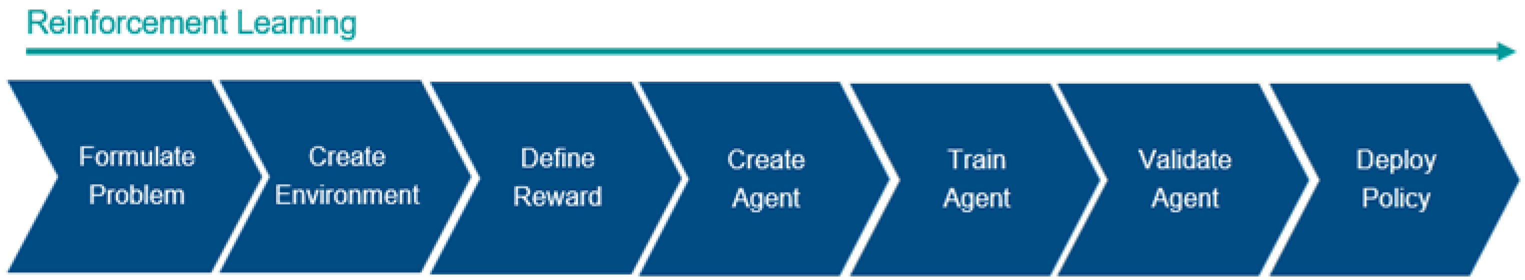
As an example, consider the task of parking a vehicle using an automated driving system. The goal of this task is for the vehicle computer (agent) to park the vehicle in the correct position and orientation. To do so, the controller uses readings from cameras, accelerometers, gyroscopes, a GPS receiver, and lidar (observations) to generate steering, braking, and acceleration commands (actions). The action commands are sent to the actuators that control the vehicle. The resulting observations depend on the actuators, sensors, vehicle dynamics, road surface, wind, and many other less-important factors. All these factors, that is, everything that is not the agent, make up the environment in reinforcement learning.

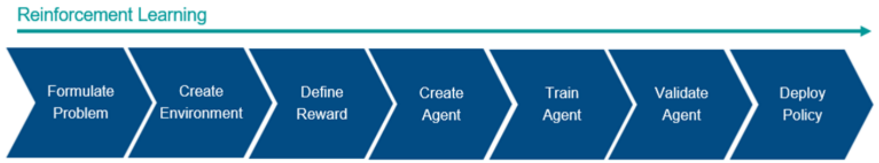
What Is Reinforcement Learning

To learn how to generate the correct actions from the observations, the computer repeatedly tries to park the vehicle using a trial-and-error process. To guide the learning process, you provide a signal that is one when the car successfully reaches the desired position and orientation and zero otherwise (reward). During each trial, the computer selects actions using a mapping (policy) initialized with some default values. After each trial, the computer updates the mapping to maximize the reward (learning algorithm). This process continues until the computer learns an optimal mapping that successfully parks the car.

Reinforcement Learning Workflow

The general workflow for training an agent using reinforcement learning includes the following steps.



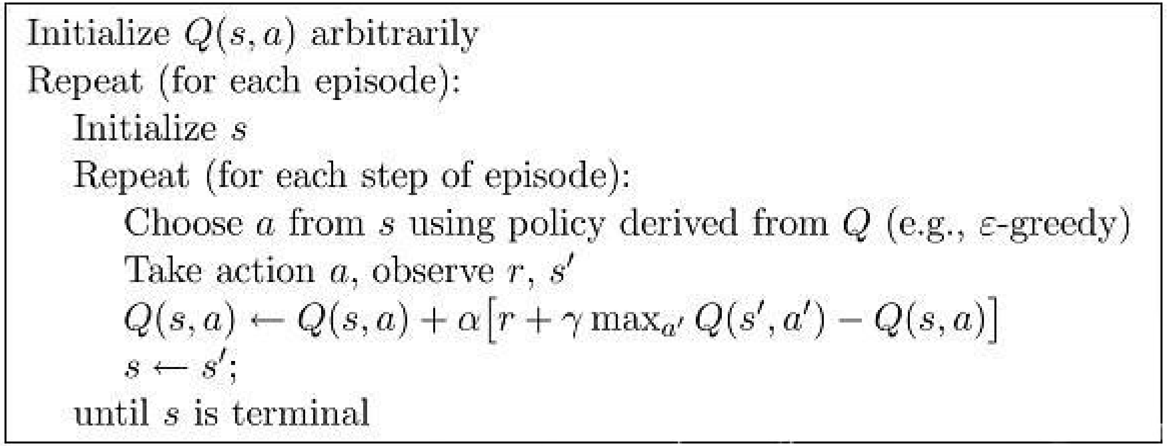


Flow

1. Formulate problem — Define the task for the agent to learn, including how the agent interacts with the environment and any primary and secondary goals the agent must achieve.
2. Create environment — Define the environment within which the agent operates, including the interface between agent and environment and the environment dynamic model.
3. Define reward — Specify the reward signal that the agent uses to measure its performance against the task goals and how to calculate this signal from the environment.
4. Create agent — Create the agent, which includes defining a policy approximator (actor) an value function approximator (critic) and configuring the agent learning algorithm.
5. Train agent — Train the agent approximators using the defined environment, reward, and agent learning algorithm.
6. Validate agent — Evaluate the performance of the trained agent by simulating the agent and environment together.
7. Deploy policy — Deploy the trained policy approximator using, for example, generated GPU code.

Q-Learning

Q-Learning 是强化学习算法中 value-based 的算法， Q 即为 Q （s， a），就是在某一个时刻的 state 状态下，采取动作 a 能够获得收益的期望，环境会根据 agent 的动作反馈相应的 reward 奖赏，所以算法的主要思想就是将 state 和 action 构建成一张 Q_table 表来存储 Q 值，然后根据 Q 值来选取能够获得最大收益的动作。

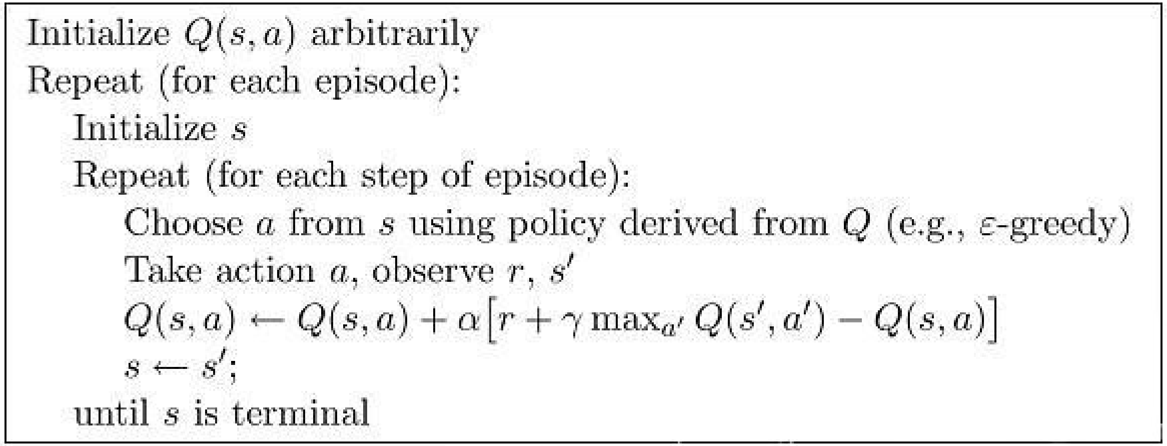


```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Repeat (for each step of episode):
        Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $a$, observe $r$, $s'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
        $s \leftarrow s'$;
    until $s$ is terminal
```

Q-Learning

Q-Learning 的目的是学习特定 state 下、特定 action 的价值。是建立一个 Q-table，以 state 为行、 action 为列，通过每个动作带来的奖赏更新 Q-table 。

Q-Learning 是 off-policy 的。异策略是指行动策略和评估策略不是一个策略。 Q-Learning 中行动策略是 ε-greedy 策略，更新 Q 表的策略是贪婪策略。

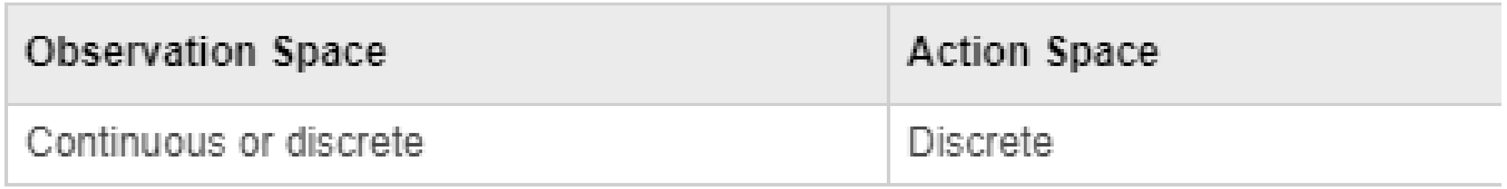


```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Repeat (for each step of episode):
        Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $a$, observe $r$, $s'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
        $s \leftarrow s'$;
    until $s$ is terminal
```

Q-Learning Agents

The Q-learning algorithm is a model-free, online, off-policy reinforcement learning method. A Q-learning agent is a value-based reinforcement learning agent that trains a critic to estimate the return or future rewards. For a given observation, the agent selects and outputs the action for which the estimated return is greatest.

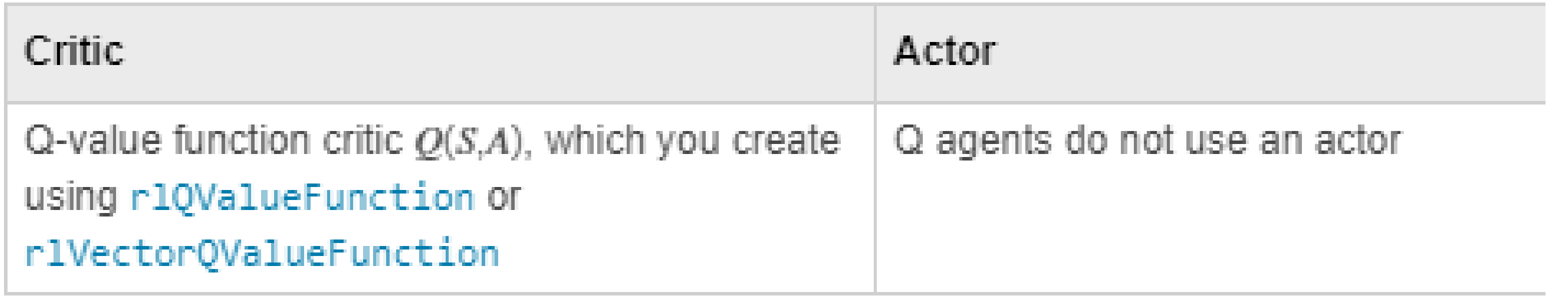
Q-learning agents can be trained in environments with the following observation and action spaces.



Observation Space	Action Space
Continuous or discrete	Discrete

Q-Learning Agents

Q agents use the following critic.



Critic	Actor
Q-value function critic $Q(S,A)$, which you create using `rlQValueFunction` or `rlVectorQValueFunction`	Q agents do not use an actor

During training, the agent explores the action space using epsilon-greedy exploration. During each control interval the agent selects a random action with probability ϵ, otherwise it selects the action for which the value function greatest with probability $1–\epsilon$.

Critic Function Approximator

To estimate the value function, a Q-learning agent maintains a critic $Q(S,A;\phi)$, which is a function approximator with parameters ϕ. The critic takes observation S and action A as inputs and returns the corresponding expectation of the long-term reward.

For critics that use table-based value functions, the parameters in ϕ are the actual $Q(S,A)$ values in the table.

During training, the agent tunes the parameter values in ϕ. After training, the parameters remain at their tuned value and the trained value function approximator is stored in critic $Q(S,A)$.

Agent Creation

To create a Q-learning agent:

1. Create a critic using an rlQValueFunction object.

2. Specify agent options using an rlQAgentOptions object.

3. Create the agent using an rlQAgent object.

Training Algorithm

Q-learning agents use the following training algorithm. To configure the training algorithm, specify options using an rlQAgentOptions object.

Initialize the critic $Q(S,A;\phi)$ with random parameter values in ϕ.

For each training episode:

1 Get the initial observation S from the environment.

Training Algorithm

2. Repeat the following for each step of the episode until S is a terminal state.

   a. For the current observation S, select a random action A with probability ϵ. Otherwise, select the action for which the critic value function is greatest.

   $$A = \arg\max_{A} Q(S, A; \phi)$$

   To specify ϵ and its decay rate, use the `EpsilonGreedyExploration` option.

   b. Execute action A. Observe the reward R and next observation S'.

   c. If S' is a terminal state, set the value function target y to R. Otherwise, set it to

   $$y = R + \gamma \max_{A} Q(S', A; \phi)$$

   To set the discount factor γ, use the `DiscountFactor` option.

   d. Compute the difference ΔQ between the value function target and the current $Q(S,A;\phi)$ value.

   $$\Delta Q = y - Q(S, A; \phi)$$

Training Algorithm

e. Update the critic using the learning rate α. Specify the learning rate when you create the critic by setting the `LearnRate` option in the `rlCriticOptimizerOptions` property within the agent options object.

- For table-based critics, update the corresponding $Q(S,A)$ value in the table.

$$Q(S, A) = Q(S, A; \phi) + \alpha \cdot \Delta Q$$

- For all other types of critics, compute the gradients $\Delta\phi$ of the loss function with respect to the parameters ϕ. Then, update the parameters based on the computed gradients. In this case, the loss function is the square of ΔQ.

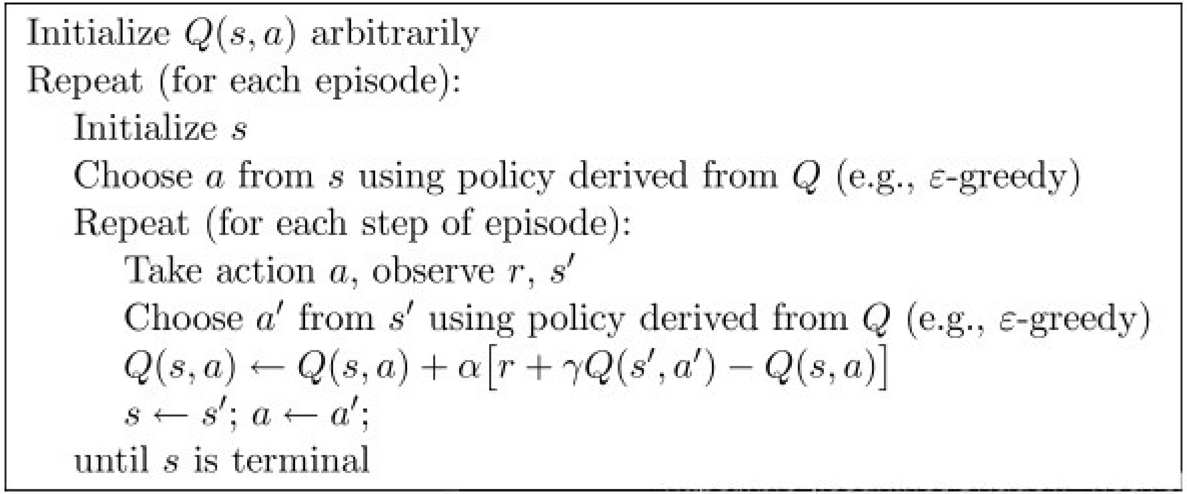
$$\Delta\phi = \frac{1}{2}\nabla_{\phi}(\Delta Q)^2$$

$$\phi = \phi + \alpha \cdot \Delta\phi$$

f. Set the observation S to S'.

SARSA

Sarsa 全称是 state-action-reward-state'-action' 。 也是采用 Q-table 的方式存储动作值函数；而且决策部分和 Q-Learning 是一样的，也是采用 ε-greedy 策略。不同的地方在于 Sarsa 的更新方式是不一样的， Sarsa 是 on-policy 的更新方式，它的行动策略和评估策略都是 ε-greedy 策略。

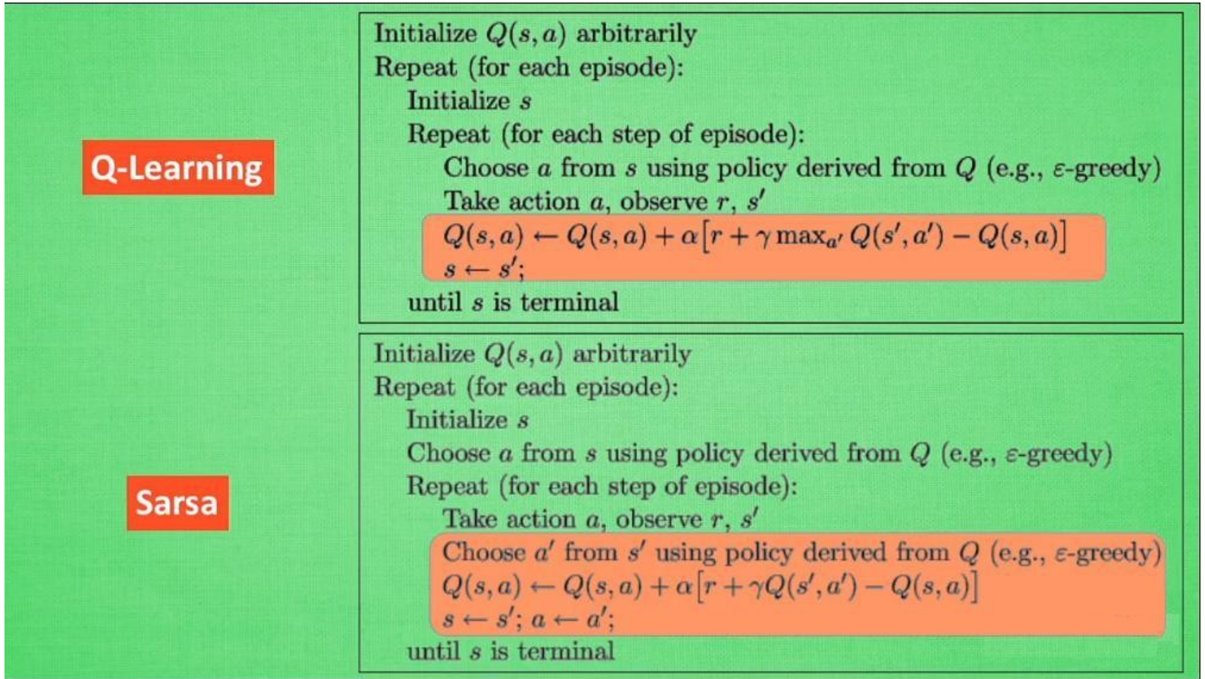


```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Repeat (for each step of episode):
        Take action $a$, observe $r$, $s'$
        Choose $a'$ from $s'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma Q(s',a') - Q(s,a)\right]$
        $s \leftarrow s'$; $a \leftarrow a'$;
    until $s$ is terminal
```

SARSA

Q-Learning vs Sarsa

- Q-Learning 算法，先假设下一步选取最大奖赏的动作，更新值函数。然后再通过 ε-greedy 策略选择动作
- Sarsa 算法，先通过 ε-greedy 策略执行动作，然后根据所执行的动作，更新值函数



Q-Learning

```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Repeat (for each step of episode):
        Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $a$, observe $r$, $s'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
        $s \leftarrow s'$;
    until $s$ is terminal
```

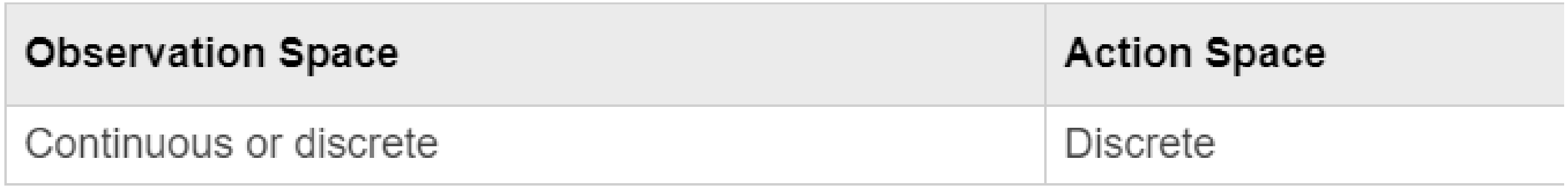
Sarsa

```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    Initialize $s$
    Choose $a$ from $s$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Repeat (for each step of episode):
        Take action $a$, observe $r$, $s'$
        Choose $a'$ from $s'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma Q(s',a') - Q(s,a)\right]$
        $s \leftarrow s'$; $a \leftarrow a'$;
    until $s$ is terminal
```

SARSA Agents

The SARSA algorithm is a model-free, online, on-policy reinforcement learning method. A SARSA agent is a value-based reinforcement learning agent that trains a critic to estimate the return or future rewards. For a given observation, the agent selects and outputs the action for which the estimated return is greatest.

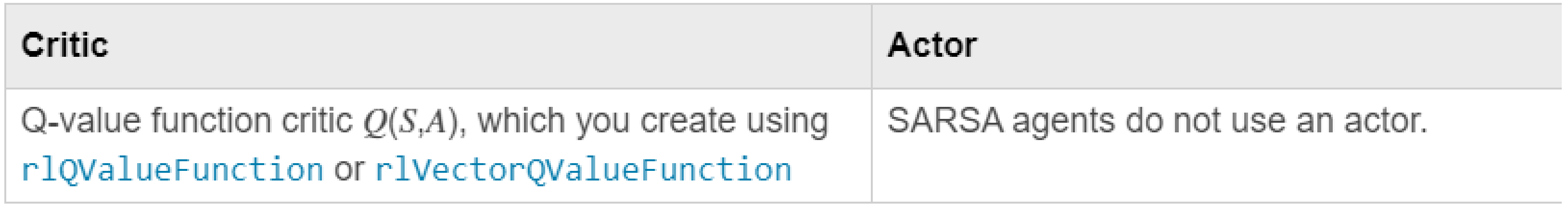
SARSA agents can be trained in environments with the following observation and action spaces.



Observation Space	Action Space
Continuous or discrete	Discrete

SARSA Agents

SARSA agents use the following critic.



Critic	Actor
Q-value function critic $Q(S,A)$, which you create using `rlQValueFunction` or `rlVectorQValueFunction`	SARSA agents do not use an actor.

During training, the agent explores the action space using epsilon-greedy exploration. During each control interval the agent selects a random action with probability ϵ, otherwise it selects the action for which the value function greatest with probability $1-\epsilon$.

Critic Function Approximator

To estimate the value function, a SARSA agent maintains a critic Q(S,A;ϕ), which is a function approximator with parameters ϕ. The critic takes observation S and action A as inputs and returns the corresponding expectation of the long-term reward.

For critics that use table-based value functions, the parameters in ϕ are the actual Q(S,A) values in the table.

During training, the agent tunes the parameter values in ϕ. After training, the parameters remain at their tuned value and the trained value function approximator is stored in critic Q(S,A).

Agent Creation

To create a SARSA agent:

1. Create a critic using an rlQValueFunction object.

2. Specify agent options using an rlSARSAAgentOptions object.

3. Create the agent using an rlSARSAAgent object.

Training Algorithm

SARSA agents use the following training algorithm. To configure the training algorithm, specify options using an rlSARSAAgentOptions object.

Initialize the critic Q(S,A;ϕ) with random parameter values in ϕ.

For each training episode:

1. Get the initial observation S from the environment.

2. For the current observation S, select a random action A with probability ϵ. Otherwise, select the action for which the critic value function is greatest.

$$A = \arg\max_{A} Q(S, A; \phi)$$

Training Algorithm

3. Repeat the following for each step of the episode until S is a terminal state:

   a. Execute action A_0. Observe the reward R and next observation S'.

   b. For the current observation S', select a random action A' with probability ϵ. Otherwise, select the action for which the critic value function is greatest.

   $$A' = \arg\max_{A'} Q(S', A'; \phi)$$

   c. If S' is a terminal state, set the value function target y to R. Otherwise, set it to

   $$y = R + \gamma Q(S', A'; \phi)$$

   To set the discount factor γ, use the `DiscountFactor` option.

   d. Compute the difference ΔQ between the value function target and the current $Q(S,A;\phi)$ value.

   $$\Delta Q = y - Q(S, A; \phi)$$

Training Algorithm

e. Update the critic using the learning rate α. Specify the learning rate when you create the critic by setting the `LearnRate` option in the `rlCriticOptimizerOptions` property within the agent options object.

- For table-based critics, update the corresponding $Q(S,A)$ value in the table.

$$Q(S, A) = Q(S, A; \phi) + \alpha \cdot \Delta Q$$

- For all other types of critics, compute the gradients $\Delta\phi$ of the loss function with respect to the parameters ϕ. Then, update the parameters based on the computed gradients. In this case, the loss function is the square of ΔQ.

$$\Delta\phi = \frac{1}{2}\nabla_{\phi}(\Delta Q)^2$$

$$\phi = \phi + \alpha \cdot \Delta\phi$$

f. Set the observation S to S'.

g. Set the action A to A'.

DRL 示例

- Train Reinforcement Learning Agent in Basic Grid World
- Create MATLAB Environment Using Custom Functions

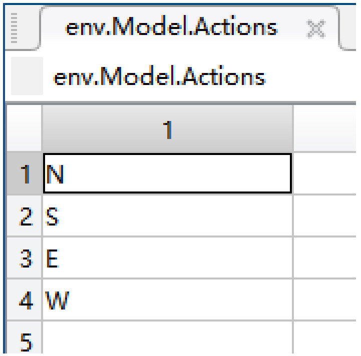
Train Reinforcement Learning Agent in Basic Grid World

This example shows how to solve a grid world environment using reinforcement learning by training Q-learning and SARSA agents. For more information on these agents, see Q-Learning Agents and SARSA Agents.

This grid world environment has the following configuration and rules:

1. The grid world is 5-by-5 and bounded by borders, with four possible actions (North = 1, South = 2, East = 3, West = 4).
2. The agent begins from cell [2,1] (second row, first column).
3. The agent receives a reward +10 if it reaches the terminal state at cell [5,5] (blue).
4. The environment contains a special jump from cell [2,4] to cell [4,4] with a reward of +5.
5. The agent is blocked by obstacles (black cells).
6. All other actions result in –1 reward.

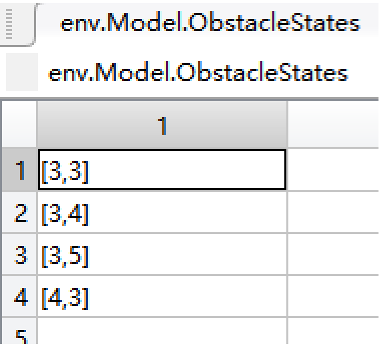
Train Reinforcement Learning Agent in Basic Grid Wor



env.Model.Actions

env.Model.Actions

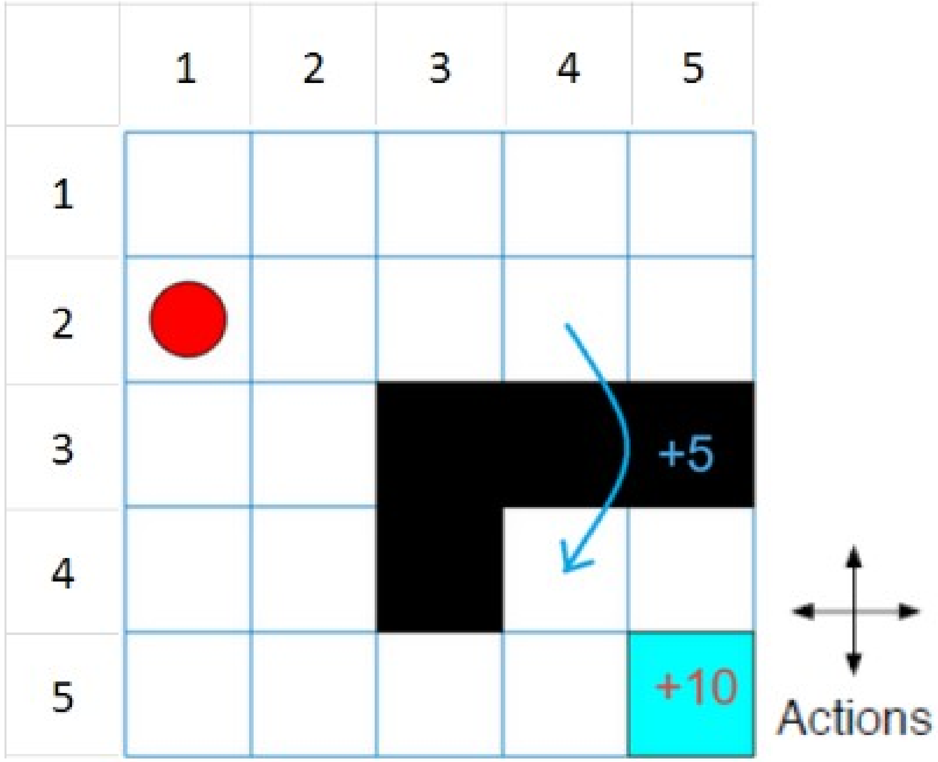
	1
1	N
2	S
3	E
4	W
5	

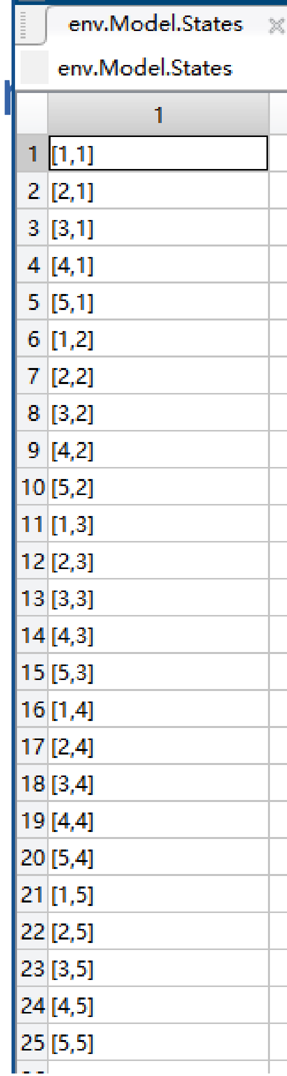


env.Model.ObstacleStates

env.Model.ObstacleStates

	1
1	[3,3]
2	[3,4]
3	[3,5]
4	[4,3]
5	





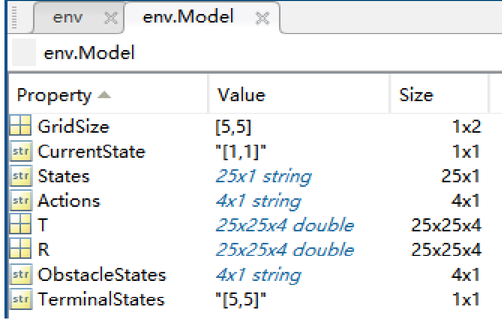
env.Model.States

env.Model.States

	1
1	[1,1]
2	[2,1]
3	[3,1]
4	[4,1]
5	[5,1]
6	[1,2]
7	[2,2]
8	[3,2]
9	[4,2]
10	[5,2]
11	[1,3]
12	[2,3]
13	[3,3]
14	[4,3]
15	[5,3]
16	[1,4]
17	[2,4]
18	[3,4]
19	[4,4]
20	[5,4]
21	[1,5]
22	[2,5]
23	[3,5]
24	[4,5]
25	[5,5]

Create Grid World Environment

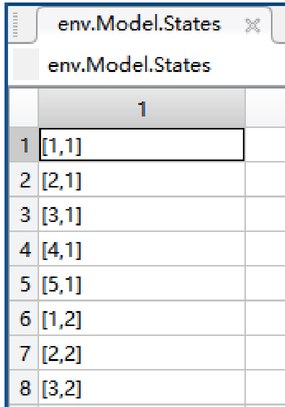
Create the basic grid world environment.
env = rlPredefinedEnv("BasicGridWorld");


env | env.Model

env.Model

Property	Value	Size
GridSize	[5,5]	1x2
CurrentState	"[1,1]"	1x1
States	25x1 string	25x1
Actions	4x1 string	4x1
T	25x25x4 double	25x25x4
R	25x25x4 double	25x25x4
ObstacleStates	4x1 string	4x1
TerminalStates	"[5,5]"	1x1

To specify that the initial state of the agent is always [2,1], create a reset function that returns the state number for the initial agent state. This function is called at the start of each training episode and simulation. States are numbered starting at position [1,1]. The state number increases as you move down the first column and then down each subsequent column. Therefore, create an anonymous function handle that sets the initial state to 2.
env.ResetFcn = @() 2;


env.Model.States

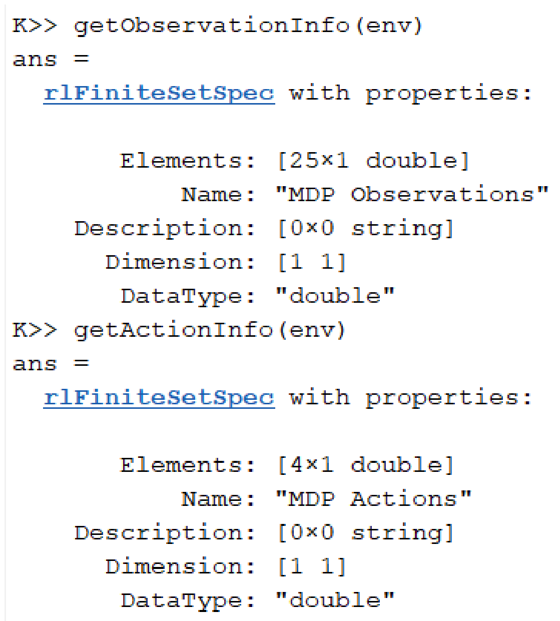
env.Model.States

	1
1	[1,1]
2	[2,1]
3	[3,1]
4	[4,1]
5	[5,1]
6	[1,2]
7	[2,2]
8	[3,2]

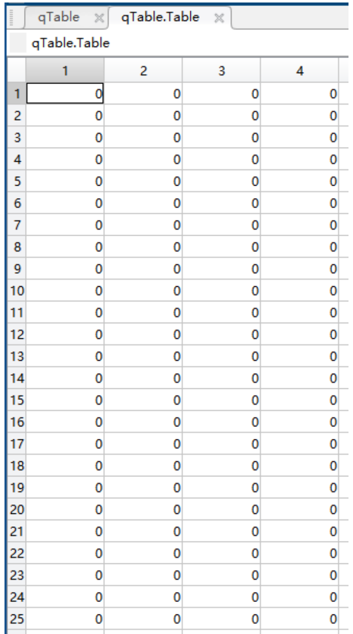
Fix the random generator seed for reproducibility.
rng(0)

Create Q-Learning Agent

To create a Q-learning agent, first create a Q table using the observation and action specifications from the grid world environment. Set the learning rate of the optimizer to 0.01.
qTable = rlTable(getObservationInfo(env),getActionInfo(env));



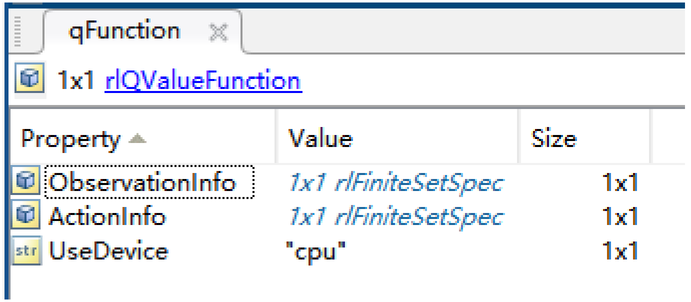
```
K>> getObservationInfo(env)
ans =
  rlFiniteSetSpec with properties:

       Elements: [25×1 double]
           Name: "MDP Observations"
    Description: [0×0 string]
      Dimension: [1 1]
       DataType: "double"
K>> getActionInfo(env)
ans =
  rlFiniteSetSpec with properties:

       Elements: [4×1 double]
           Name: "MDP Actions"
    Description: [0×0 string]
      Dimension: [1 1]
       DataType: "double"
```



	1	2	3	4
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0
9	0	0	0	0
10	0	0	0	0
11	0	0	0	0
12	0	0	0	0
13	0	0	0	0
14	0	0	0	0
15	0	0	0	0
16	0	0	0	0
17	0	0	0	0
18	0	0	0	0
19	0	0	0	0
20	0	0	0	0
21	0	0	0	0
22	0	0	0	0
23	0	0	0	0
24	0	0	0	0
25	0	0	0	0

Create Q-Learning Agent

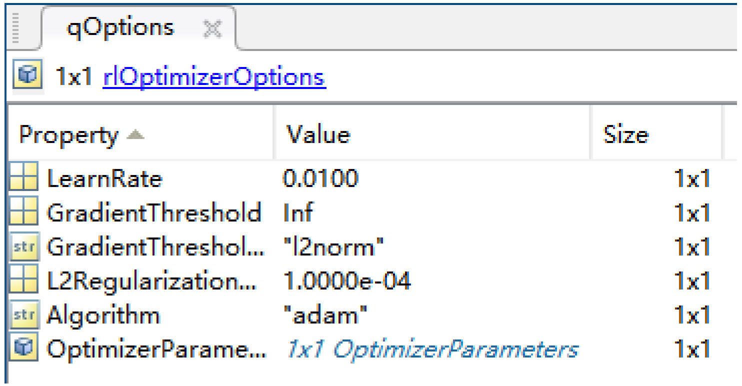
```
qFunction =
rlQValueFunction(qTable,getObservationInfo(env),getActionInfo(env));
qOptions = rlOptimizerOptions("LearnRate",0.01);
```



qFunction

1x1 rlQValueFunction

Property	Value	Size
ObservationInfo	*1x1 rlFiniteSetSpec*	1x1
ActionInfo	*1x1 rlFiniteSetSpec*	1x1
UseDevice	"cpu"	1x1



qOptions

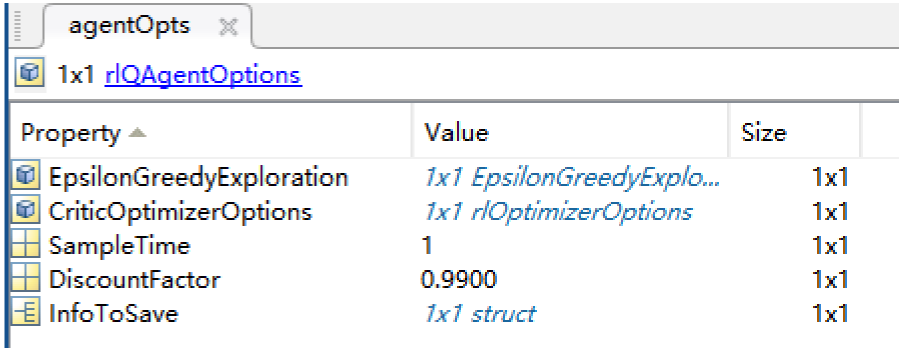
1x1 rlOptimizerOptions

Property	Value	Size
LearnRate	0.0100	1x1
GradientThreshold	Inf	1x1
GradientThreshol...	"l2norm"	1x1
L2Regularization...	1.0000e-04	1x1
Algorithm	"adam"	1x1
OptimizerParame...	*1x1 OptimizerParameters*	1x1

Create Q-Learning Agent

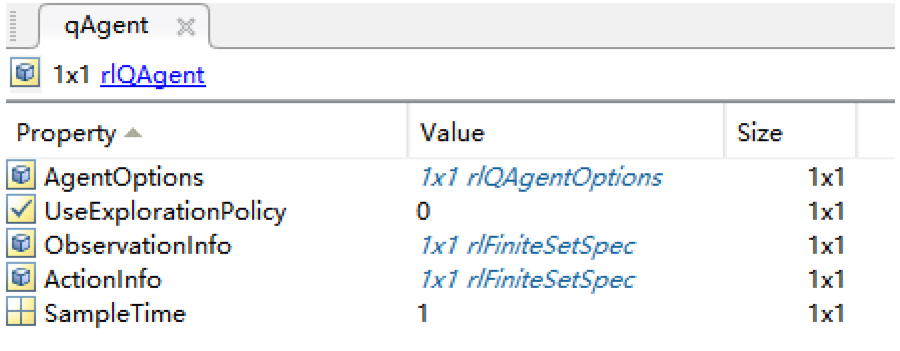
Next, create a Q-learning agent using the Q value function and configure the epsilon-greedy exploration.

```
agentOpts = rlQAgentOptions;
agentOpts.EpsilonGreedyExploration.Epsilon = .04;
agentOpts.CriticOptimizerOptions = qOptions;
qAgent = rlQAgent(qFunction,agentOpts);
```


agentOpts

1x1 rlQAgentOptions

Property	Value	Size
EpsilonGreedyExploration	*1x1 EpsilonGreedyExplo...*	1x1
CriticOptimizerOptions	*1x1 rlOptimizerOptions*	1x1
SampleTime	1	1x1
DiscountFactor	0.9900	1x1
InfoToSave	*1x1 struct*	1x1


qAgent

1x1 rlQAgent

Property	Value	Size
AgentOptions	*1x1 rlQAgentOptions*	1x1
UseExplorationPolicy	0	1x1
ObservationInfo	*1x1 rlFiniteSetSpec*	1x1
ActionInfo	*1x1 rlFiniteSetSpec*	1x1
SampleTime	1	1x1

Train Q-Learning Agent

To train the agent, first specify the training options. For this example, use the following options:
Train for at most 200 episodes. Specify that each episode lasts for most 50 time steps.
Stop training when the agent receives an average cumulative reward greater than 10 over 30 consecutive episodes.

```
trainOpts = rlTrainingOptions;
trainOpts.MaxStepsPerEpisode = 50;
trainOpts.MaxEpisodes= 200;
trainOpts.StopTrainingCriteria = "AverageReward";
trainOpts.StopTrainingValue = 11;
trainOpts.ScoreAveragingWindowLength = 30;
```

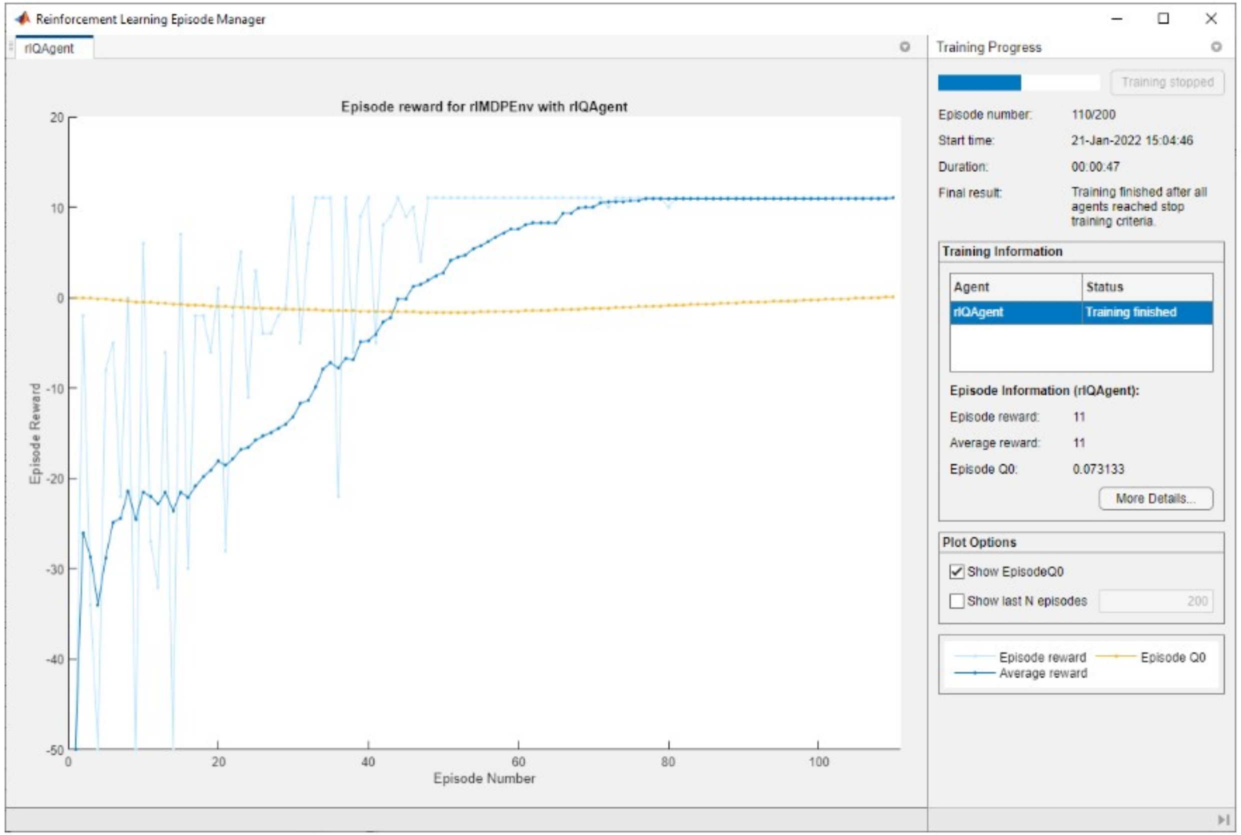
Train Q-Learning Agent

Train the Q-learning agent using the train function. Training can take several minutes to complete. To save time while running this example, load a pretrained agent by setting doTraining to false. To train the agent yourself, set doTraining to true.

```
doTraining = false;

if doTraining
   % Train the agent.
   trainingStats = train(qAgent,env,trainOpts);
else
   % Load the pretrained agent for the example.
   load('basicGWQAgent.mat','qAgent')
end
```

Train Q-Learning Agent

The Episode Manager window opens and displays the training progress.



Episode number: 110/200
Start time: 21-Jan-2022 15:04:46
Duration: 00:00:47
Final result: Training finished after all agents reached stop training criteria.

Training Information

Agent	Status
rlQAgent	Training finished

Episode Information (rlQAgent):

Episode reward: 11
Average reward: 11
Episode Q0: 0.073133

Plot Options

[✓] Show EpisodeQ0
[ ] Show last N episodes 200

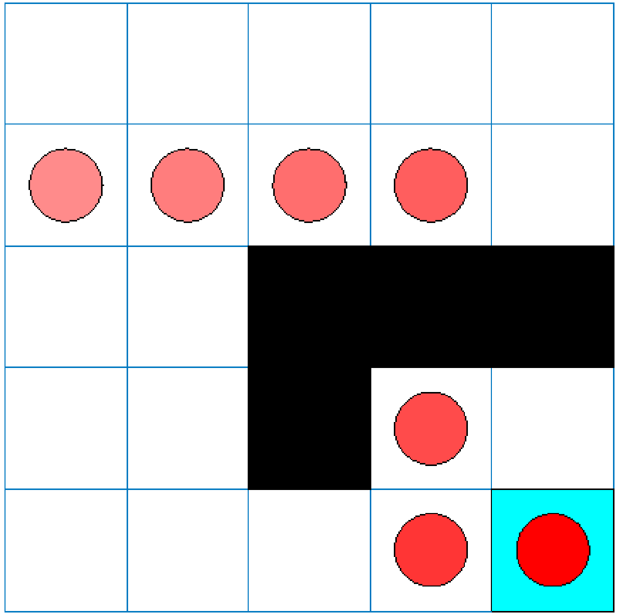
Validate Q-Learning Results

To validate the training results, simulate the agent in the training environment.
Before running the simulation, visualize the environment and configure the visualization to maintain a trace of the agent states.

```
plot(env)
env.Model.Viewer.ShowTrace = true;
env.Model.Viewer.clearTrace;
```

Validate Q-Learning Results

Simulate the agent in the environment using the sim function.
sim(qAgent,env)
The agent trace shows that the agent successfully finds the jump from cell [2,4] to cell [4,4].



Create and Train SARSA Agent

To create a SARSA agent, use the same Q value function and epsilon-greedy configuration as for the Q-learning agent.

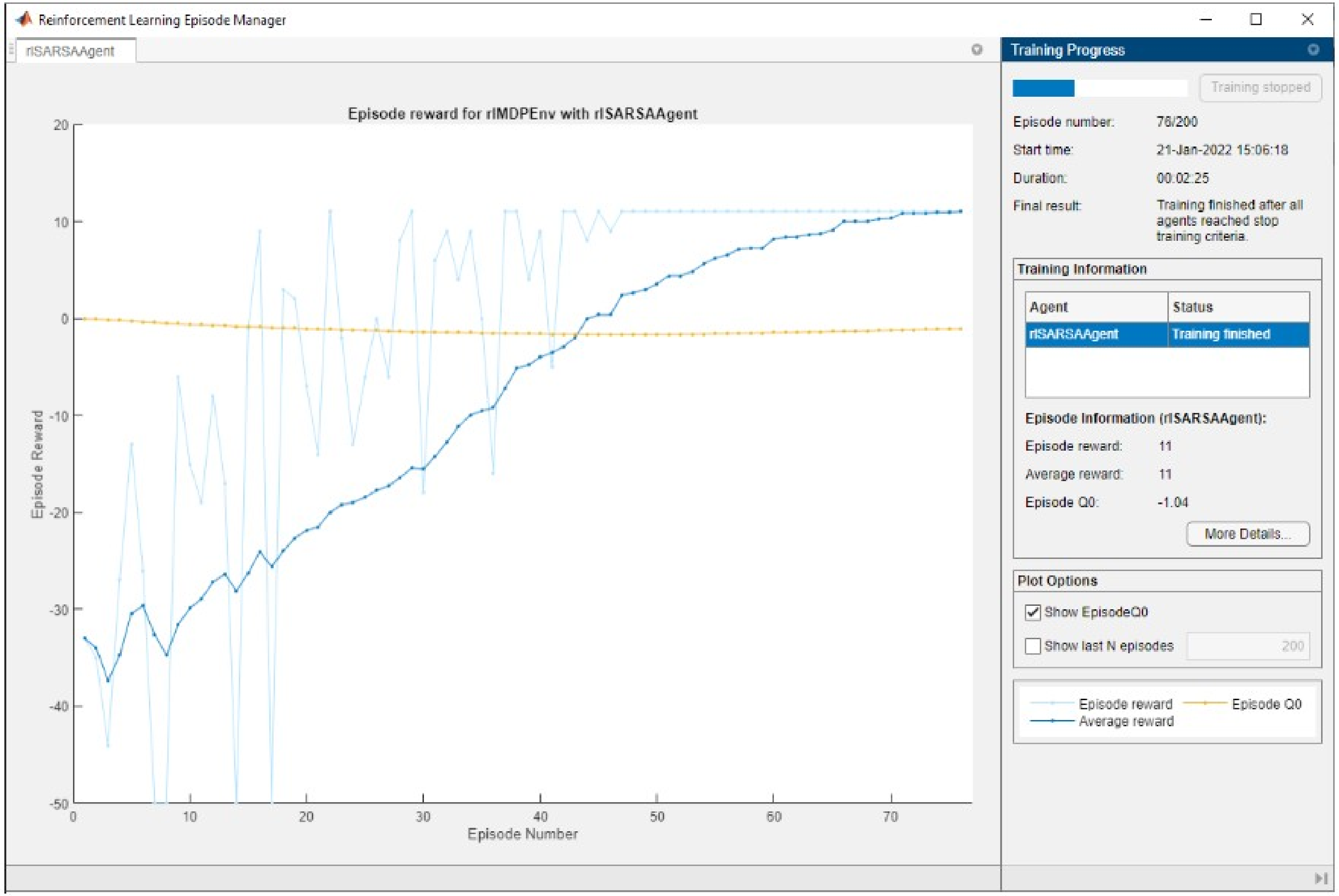
```
agentOpts = rlSARSAAgentOptions;
agentOpts.EpsilonGreedyExploration.Epsilon = 0.04;
agentOpts.CriticOptimizerOptions = qOptions;
sarsaAgent = rlSARSAAgent(qFunction,agentOpts);
```

Create and Train SARSA Agent

Train the SARSA agent using the train function. Training can take several minutes to complete. To save time while running this example, load a pretrained agent by setting doTraining to false. To train the agent yourself, set doTraining to true.

```
doTraining = false;

if doTraining
   % Train the agent.
   trainingStats = train(sarsaAgent,env,trainOpts);
else
   % Load the pretrained agent for the example.
   load('basicGWSarsaAgent.mat','sarsaAgent')
end
```

Create and Train SARSA Agent



Episode number: 76/200

Start time: 21-Jan-2022 15:06:18

Duration: 00:02:25

Final result: Training finished after all agents reached stop training criteria.

Training Information

Agent	Status
rlSARSAAgent	Training finished

Episode Information (rlSARSAAgent):

Episode reward: 11

Average reward: 11

Episode Q0: -1.04

Plot Options

[✓] Show EpisodeQ0

[ ] Show last N episodes 200

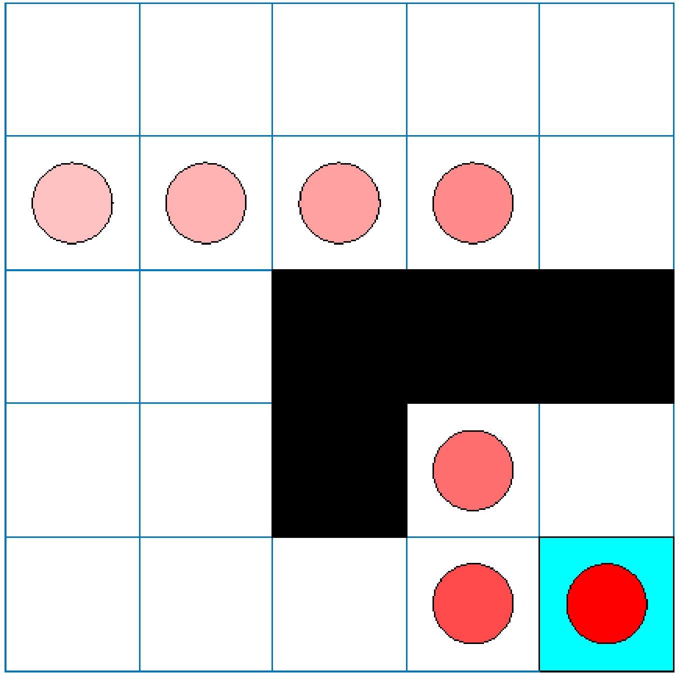
Validate SARSA Training

To validate the training results, simulate the agent in the training environment.

```
plot(env)
env.Model.Viewer.ShowTrace = true;
env.Model.Viewer.clearTrace;
```

Validate SARSA Training

Simulate the agent in the environment.
sim(sarsaAgent,env)
The SARSA agent finds the same grid world solution as the Q-learning agent.

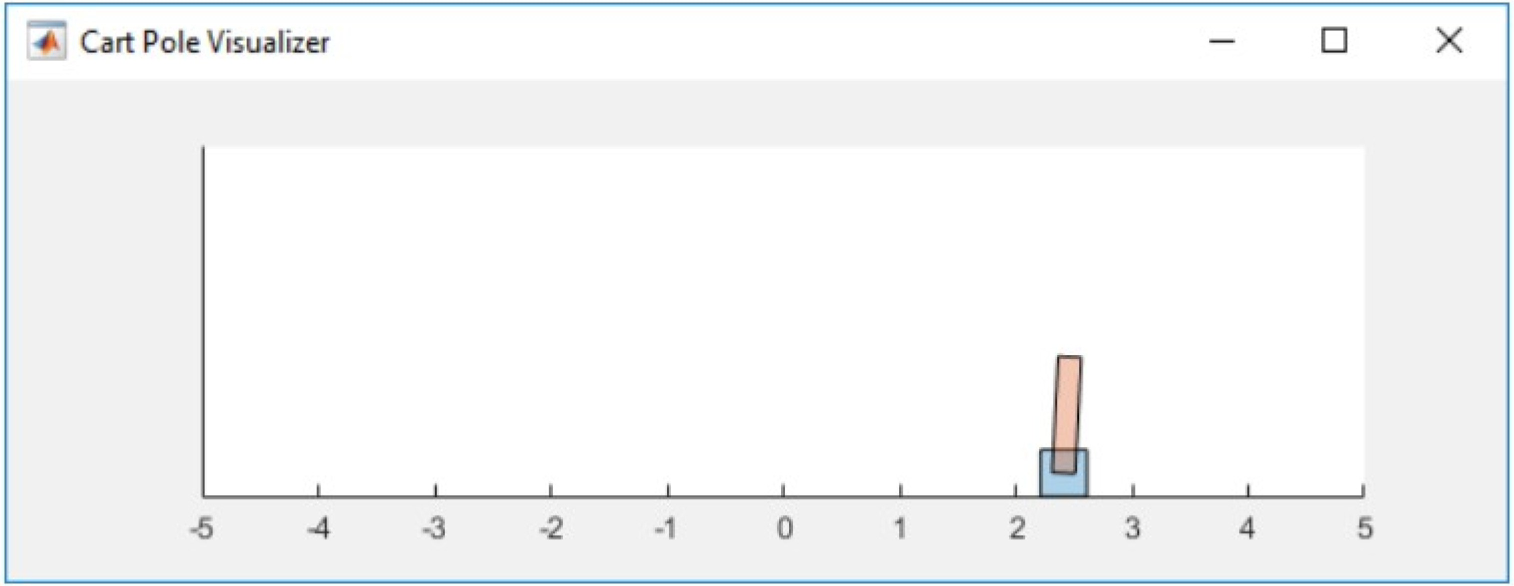


Create MATLAB Environment Using Custom Functions

This example shows how to create a cart-pole environment by supplying custom dynamic functions in MATLAB®.
Using the rlFunctionEnv function, you can create a MATLAB reinforcement learning environment from an observation specification, an action specification, and user-defined step and reset functions. You can then train a reinforcement learning agent in this environment. The necessary step and reset functions are already defined for this example.
Creating an environment using custom functions is useful for environments with less complex dynamics, environments with no special visualization requirements, or environments with interfaces to third-party libraries. For more complex environments, you can create an environment object using a template class.

Cart-Pole MATLAB Environment

The cart-pole environment is a pole attached to an unactuated joint on a cart, which moves along a frictionless track. The training goal is to make the pendulum stand upright without falling over.



Cart-Pole MATLAB Environment

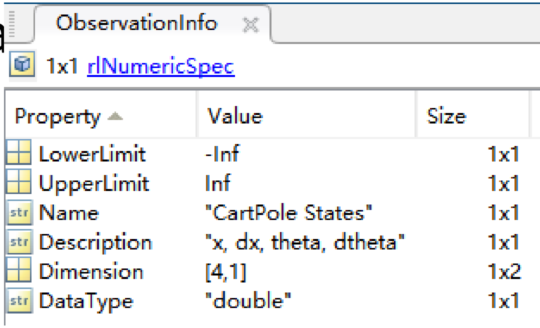
For this environment:

- The upward balanced pendulum position is 0 radians, and the downward hanging position is pi radians.
- The pendulum starts upright with an initial angle that is between –0.05 and 0.05.
- The force action signal from the agent to the environment is from –10 to 10 N.
- The observations from the environment are the cart position, cart velocity, pendulum angle, and pendulum angle derivative.
- The episode terminates if the pole is more than 12 degrees from vertical, or if the cart moves more than 2.4 m from the original position.
- A reward of +1 is provided for every time step that the pole remains upright. A penalty of –10 is applied when the pendulum falls.

Observation and Action Specifications

The observations from the environment are the cart position, cart velocity, pendulum angle, and pendulum angle deriva

```
ObservationInfo = rlNumericSpec([4 1]);
ObservationInfo.Name = 'CartPole States';
ObservationInfo.Description = 'x, dx, theta, dtheta';
```



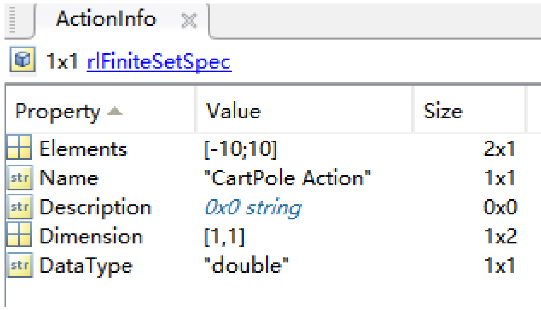
ObservationInfo

1x1 rlNumericSpec

Property	Value	Size
LowerLimit	-Inf	1x1
UpperLimit	Inf	1x1
Name	"CartPole States"	1x1
Description	"x, dx, theta, dtheta"	1x1
Dimension	[4,1]	1x2
DataType	"double"	1x1

The environment has a discrete action space where the agent can apply one of two possible force values to the cart: -10 or 10 N.

```
ActionInfo = rlFiniteSetSpec([-10 10]);
ActionInfo.Name = 'CartPole Action';
```



ActionInfo

1x1 rlFiniteSetSpec

Property	Value	Size
Elements	[-10;10]	2x1
Name	"CartPole Action"	1x1
Description	*0x0 string*	0x0
Dimension	[1,1]	1x2
DataType	"double"	1x1

Create Environment Using Function Names

To define a custom environment, first specify the custom step and reset functions. These functions must be in your current working folder or on the MATLAB path.
The custom reset function sets the default state of the environment. This function must have the following signature.

```
[InitialObservation,LoggedSignals] = myResetFunction()
```

To pass information from one step to the next, such as the environment state, use LoggedSignals. For this example, LoggedSignals contains the states of the cart-pole environment: the position and velocity of the cart, the pendulum angle, and the pendulum angle derivative. The reset function sets the cart angle to a random value each time the environment is reset.

Create Environment Using Function Names

For this example, use the custom reset function defined in myResetFunction.m.

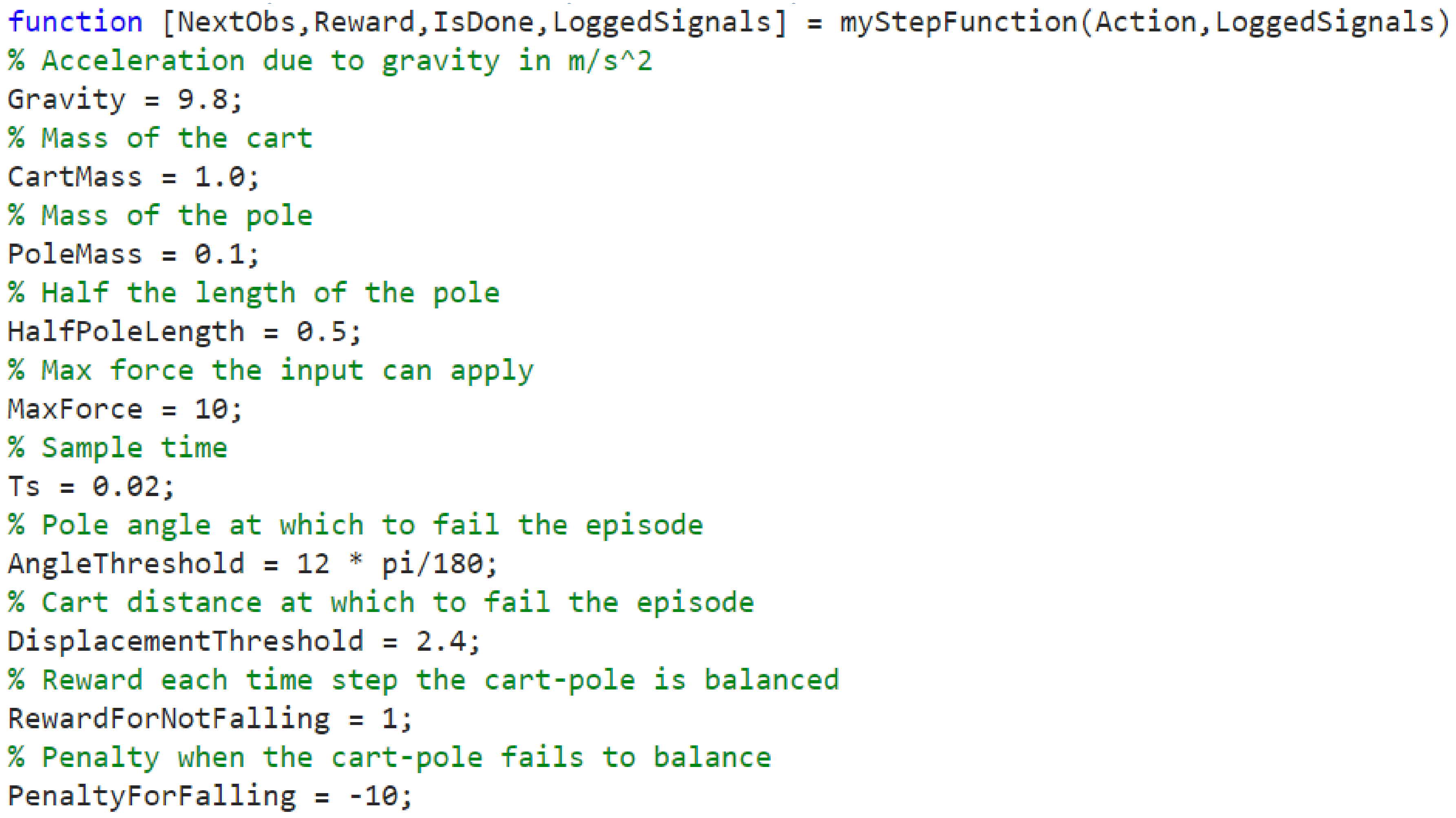
```
function [InitialObservation, LoggedSignal] = myResetFunction()
% Reset function to place custom cart-pole environment into a random
% initial state.

% Theta (randomize)
T0 = 2 * 0.05 * rand() - 0.05;
% Thetadot
Td0 = 0;
% X
X0 = 0;
% Xdot
Xd0 = 0;

% Return initial environment state variables as logged signals.
LoggedSignal.State = [X0;Xd0;T0;Td0];
InitialObservation = LoggedSignal.State;

end
```

Create Environment Using Function Names

The custom step function specifies how the environment advances to the next state based on a given action. This function must have the following signature.

```
[Observation,Reward,IsDone,LoggedSignals] =
myStepFunction(Action,LoggedSignals)
```

To get the new state, the environment applies the dynamic equation to the current state stored in LoggedSignals, which is similar to giving an initial condition to a differential equation. The new state is stored in LoggedSignals and returned as an output.
For this example, use the custom step function defined in myStepFunction.m. For implementation simplicity, this function redefines physical constants, such as the cart mass, every time step is executed



```
function [NextObs,Reward,IsDone,LoggedSignals] = myStepFunction(Action,LoggedSignals)
% Acceleration due to gravity in m/s^2
Gravity = 9.8;
% Mass of the cart
CartMass = 1.0;
% Mass of the pole
PoleMass = 0.1;
% Half the length of the pole
HalfPoleLength = 0.5;
% Max force the input can apply
MaxForce = 10;
% Sample time
Ts = 0.02;
% Pole angle at which to fail the episode
AngleThreshold = 12 * pi/180;
% Cart distance at which to fail the episode
DisplacementThreshold = 2.4;
% Reward each time step the cart-pole is balanced
RewardForNotFalling = 1;
% Penalty when the cart-pole fails to balance
PenaltyForFalling = -10;
```

```matlab
% Check if the given action is valid.
if ~ismember(Action,[-MaxForce MaxForce])
    error('Action must be %g for going left and %g for going right.',...
        -MaxForce,MaxForce);
end
Force = Action;

% Unpack the state vector from the logged signals.
State = LoggedSignals.State;
XDot = State(2);
Theta = State(3);
ThetaDot = State(4);

% Cache to avoid recomputation.
CosTheta = cos(Theta);
SinTheta = sin(Theta);
SystemMass = CartMass + PoleMass;
temp = (Force + PoleMass*HalfPoleLength*ThetaDot*ThetaDot*SinTheta)/SystemMass;
```

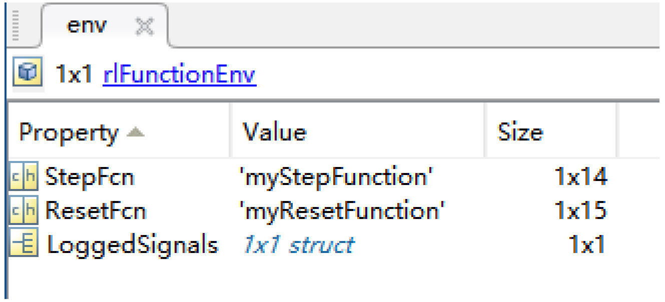
```
% Apply motion equations.
ThetaDotDot = (Gravity*SinTheta - CosTheta*temp) / ...
    (HalfPoleLength*(4.0/3.0 - PoleMass*CosTheta*CosTheta/SystemMass));
XDotDot  = temp - PoleMass*HalfPoleLength*ThetaDotDot*CosTheta/SystemMass;

% Perform Euler integration.
LoggedSignals.State = State + Ts.*[XDot;XDotDot;ThetaDot;ThetaDotDot];

% Transform state to observation.
NextObs = LoggedSignals.State;

% Check terminal condition.
X = NextObs(1);
Theta = NextObs(3);
IsDone = abs(X) > DisplacementThreshold || abs(Theta) > AngleThreshold;

% Get reward.
if ~IsDone
    Reward = RewardForNotFalling;
else
    Reward = PenaltyForFalling;
end

end
```

Create Environment Using Function Names

Construct the custom environment using the defined observation specification, action specification, and function names.

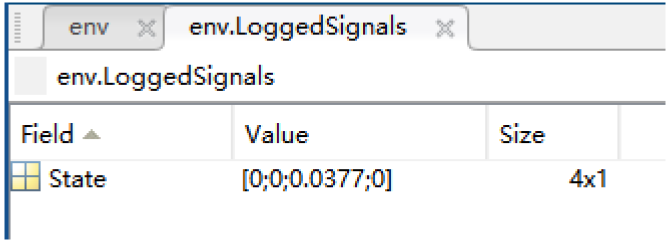
```
env = rlFunctionEnv(ObservationInfo,ActionInfo,'myStepFunction','myResetFunction');
```

To verify the operation of your environment, rlFunctionEnv automatically calls validateEnvironment after creating the environment.



1x1 rlFunctionEnv

Property	Value	Size
StepFcn	'myStepFunction'	1x14
ResetFcn	'myResetFunction'	1x15
LoggedSignals	*1x1 struct*	1x1



env.LoggedSignals

Field	Value	Size
State	[0;0;0.0377;0]	4x1

Create Environment Using Function Handles

You can also define custom functions that have additional input arguments beyond the minimum required set. For example, to pass the additional arguments arg1 and arg2 to both the step and rest function, use the following code.

```
[InitialObservation,LoggedSignals] = myResetFunction(arg1,arg2)
[Observation,Reward,IsDone,LoggedSignals] =
myStepFunction(Action,LoggedSignals,arg1,arg2)
```

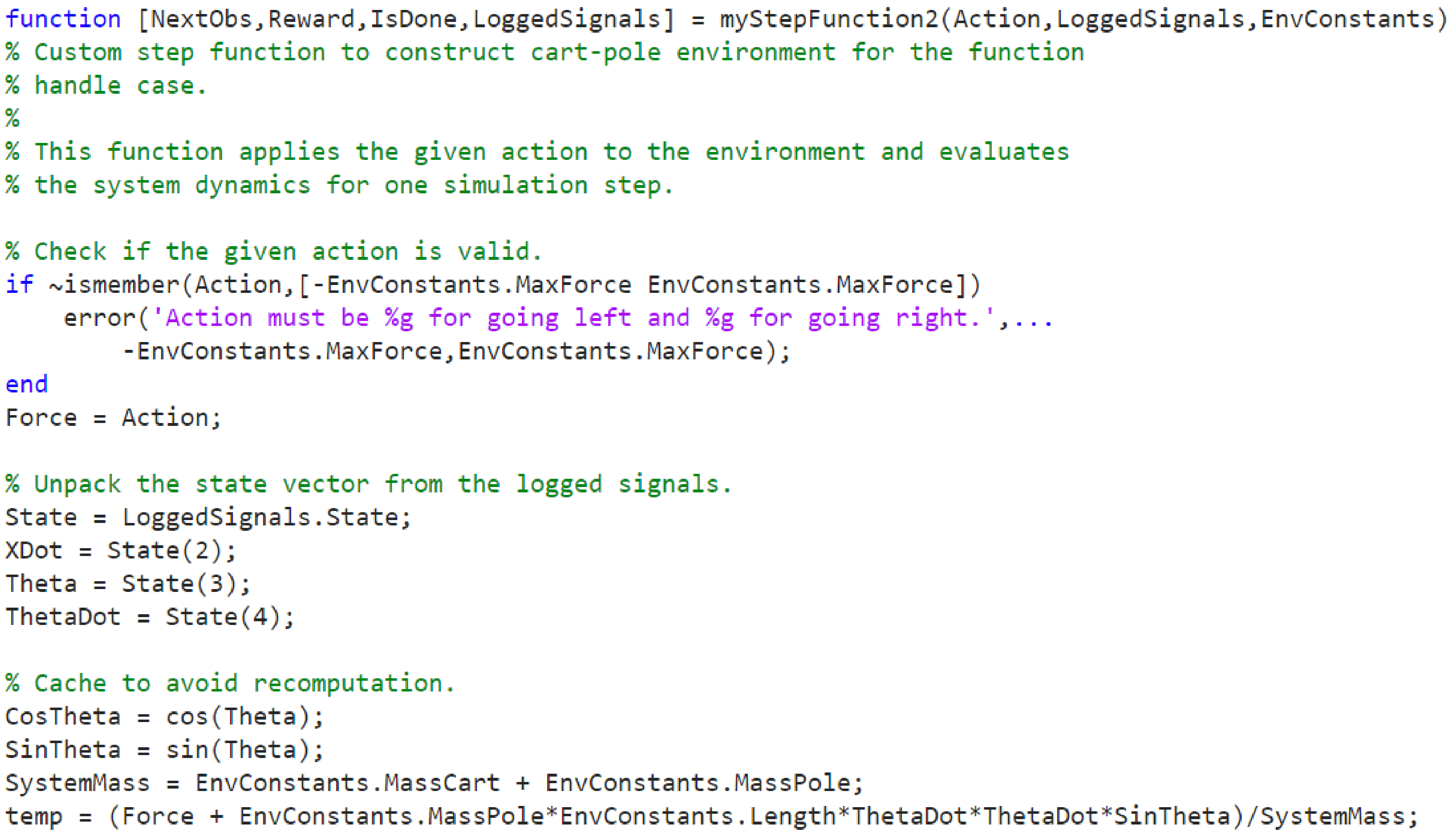
To use these functions with rlFunctionEnv, you must use anonymous function handles.

```
ResetHandle = @()myResetFunction(arg1,arg2);
StepHandle = @(Action,LoggedSignals)
myStepFunction(Action,LoggedSignals,arg1,arg2);
```

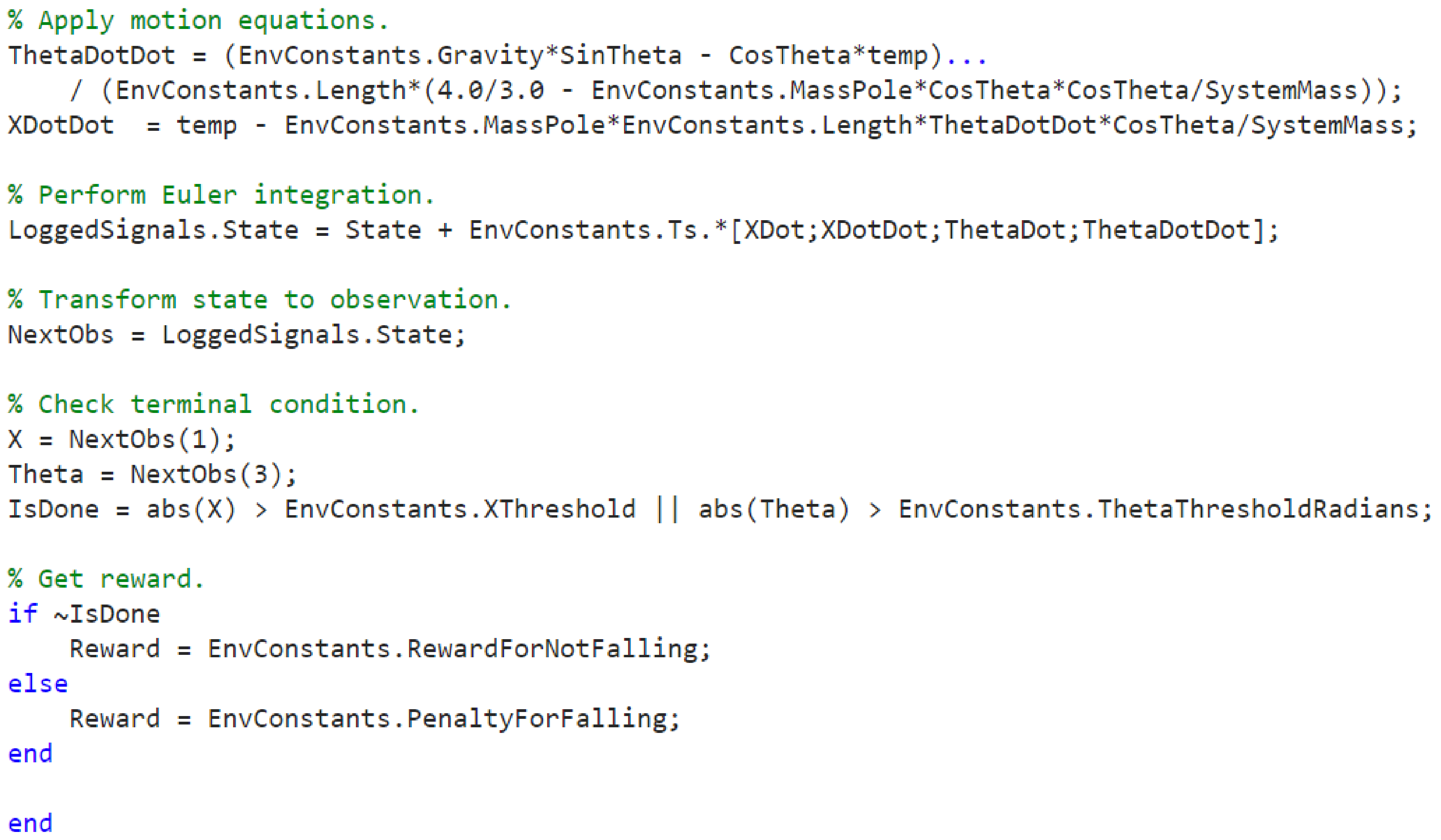
Create Environment Using Function Handles

Using additional input arguments can create a more efficient environment implementation. For example, myStepFunction2.m contains a custom step function that takes the environment constants as an input argument (envConstants). By doing so, this function avoids redefining the environment constants at each step.

```
type myStepFunction2.m
```



```matlab
function [NextObs,Reward,IsDone,LoggedSignals] = myStepFunction2(Action,LoggedSignals,EnvConstants)
% Custom step function to construct cart-pole environment for the function
% handle case.
%
% This function applies the given action to the environment and evaluates
% the system dynamics for one simulation step.

% Check if the given action is valid.
if ~ismember(Action,[-EnvConstants.MaxForce EnvConstants.MaxForce])
    error('Action must be %g for going left and %g for going right.',...
        -EnvConstants.MaxForce,EnvConstants.MaxForce);
end
Force = Action;

% Unpack the state vector from the logged signals.
State = LoggedSignals.State;
XDot = State(2);
Theta = State(3);
ThetaDot = State(4);

% Cache to avoid recomputation.
CosTheta = cos(Theta);
SinTheta = sin(Theta);
SystemMass = EnvConstants.MassCart + EnvConstants.MassPole;
temp = (Force + EnvConstants.MassPole*EnvConstants.Length*ThetaDot*ThetaDot*SinTheta)/SystemMass;
```



```
% Apply motion equations.
ThetaDotDot = (EnvConstants.Gravity*SinTheta - CosTheta*temp)...
    / (EnvConstants.Length*(4.0/3.0 - EnvConstants.MassPole*CosTheta*CosTheta/SystemMass));
XDotDot  = temp - EnvConstants.MassPole*EnvConstants.Length*ThetaDotDot*CosTheta/SystemMass;

% Perform Euler integration.
LoggedSignals.State = State + EnvConstants.Ts.*[XDot;XDotDot;ThetaDot;ThetaDotDot];

% Transform state to observation.
NextObs = LoggedSignals.State;

% Check terminal condition.
X = NextObs(1);
Theta = NextObs(3);
IsDone = abs(X) > EnvConstants.XThreshold || abs(Theta) > EnvConstants.ThetaThresholdRadians;

% Get reward.
if ~IsDone
    Reward = EnvConstants.RewardForNotFalling;
else
    Reward = EnvConstants.PenaltyForFalling;
end

end
```

Create Environment Using Function Handles

Create the structure that contains the environment constants.

```
% Acceleration due to gravity in m/s^2
envConstants.Gravity = 9.8;
% Mass of the cart
envConstants.MassCart = 1.0;
% Mass of the pole
envConstants.MassPole = 0.1;
% Half the length of the pole
envConstants.Length = 0.5;
% Max force the input can apply
envConstants.MaxForce = 10;
% Sample time
envConstants.Ts = 0.02;
% Angle at which to fail the episode
envConstants.ThetaThresholdRadians = 12 * pi/180;
% Distance at which to fail the episode
envConstants.XThreshold = 2.4;
% Reward each time step the cart-pole is balanced
envConstants.RewardForNotFalling = 1;
% Penalty when the cart-pole fails to balance
envConstants.PenaltyForFalling = -5;
```

Create Environment Using Function Handles

Create an anonymous function handle to the custom step function, passing envConstants as an additional input argument. Because envConstants is available at the time that StepHandle is created, the function handle includes those values. The values persist within the function handle even if you clear the variables.

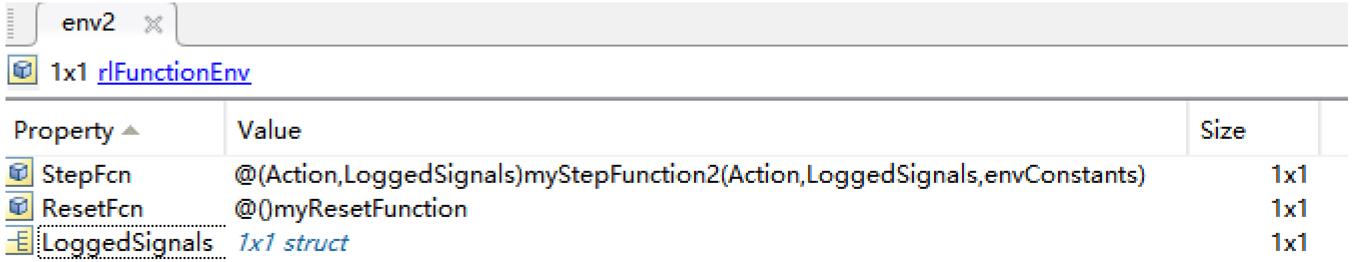
```
StepHandle = @(Action,LoggedSignals)
myStepFunction2(Action,LoggedSignals,envConstants);
```

Use the same reset function, specifying it as a function handle rather than by using its name.

```
ResetHandle = @() myResetFunction;
```

Create Environment Using Function Handles

Create the environment using the custom function handles.
env2 =
rlFunctionEnv(ObservationInfo,ActionInfo,StepHandle,ResetHandle);



env2

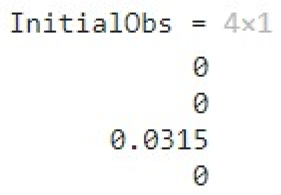
1x1 rlFunctionEnv

Property	Value	Size
StepFcn	@(Action,LoggedSignals)myStepFunction2(Action,LoggedSignals,envConstants)	1x1
ResetFcn	@()myResetFunction	1x1
LoggedSignals	*1x1 struct*	1x1

Validate Custom Functions

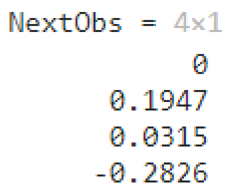
Before you train an agent in your environment, the best practice is to validate the behavior of your custom functions. To do so, you can initialize your environment using the reset function and run one simulation step using the step function. For reproducibility, set the random generator seed before validation.
Validate the environment created using function names.

```
rng(0);
InitialObs = reset(env)
```



```
InitialObs = 4×1

         0
         0
    0.0315
         0
```

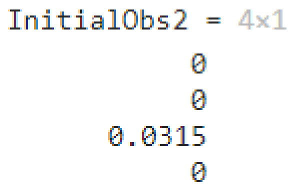
```
[NextObs,Reward,IsDone,LoggedSignals] = step(env,10);
NextObs
```



```
NextObs = 4×1

         0
    0.1947
    0.0315
   -0.2826
```

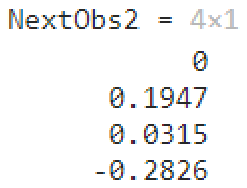
Validate Custom Functions

Validate the environment created using function handles.
rng(0);
InitialObs2 = reset(env2)



```
InitialObs2 = 4×1

          0
          0
     0.0315
          0
```

[NextObs2,Reward2,IsDone2,LoggedSignals2] = step(env2,10);
NextObs2



```
NextObs2 = 4×1

          0
     0.1947
     0.0315
    -0.2826
```

Both environments initialize and simulate successfully, producing the same state values in NextObs.

DRL 示例代码

```
%Train Reinforcement Learning Agent in Basic Grid World
openExample('rl/BasicGridWorldExample')

%Create MATLAB Environment Using Custom Functions
openExample('rl/CreateMATLABEnvironmentUsingCustomFunctionsExample')
```