Robotics Practicals

Reinforcement Learning

Wouter Caarls

Contents

Foreword

This manual is part of the Robotics Practicals lab course. In the Robotics Practicals, the aim is to familiarize yourself with the software tools that are necessary to program the robots at the Delft Biorobotics Lab (DBL). The lab courses are very much hands-on; you will be able to practice everything you learn immediately by doing the exercises that are contained in the text. Because the aim is to understand the tooling instead of a particular application, we specifically encourage you to explore and try to combine the things you've learned in new and hopefully exciting ways!

In this course, you will be using Matlab to write a complete reinforcement learning algorithm from scratch. The environment in which the robot learns will be provided for you. This manual will focus on the practical aspects of reinforcement learning, while we will be using the book "Reinforcement Learning: An Introduction" by Richard Sutton and Andrew Barto for the theory. It is freely available at `http://webdocs.cs.ualberta.ca/~sutton/book/the-book.html`. On the margin of some paragraphs we specify a section of this book you need to read before proceeding with that paragraph. For most exercises, we will also point you to the section of the book that is most helpful for solving them.

book: S&B

To pass this course, which should take roughly 40 hours of self-study, you need to write a report. This report is a protocol of all the exercises you did, so make sure you keep notes.

Enjoy!

Wouter Caarls, Delft
September 2010

Chapter 1

Introduction

book: Ch. 1

As robot tasks become more general, it is increasingly difficult to write dedicated controllers. While it is often easy to specify *what* the robot has to do, *how* to achieve that goal is not so straightforward. One way to overcome this problem is to have the robot *learn* the optimal control policy by trial and error interaction with the environment. This is the approach taken by the field of *reinforcement learning* techniques, where a policy is learnt through positive and negative reinforcements (also called *rewards*). By judiciously catalogueing the reinforcements, the robot learns to predict which actions have the highest rewards. The control policy is then simply a matter of choosing the action associated with the best prediction. Of course, reinforcement learning can be used in many areas outside robotics, but robotics is what we will focus on.

Although the goal of reinforcement learning is application in the real world, much research has focused on simulations of less or more complex environments. This is simply because the real world provides many distractions which have to be solved before the algorithmic aspects can be addressed. We will therefore also restrict ourselves to a simulation. However, our main simulator has the advantage that it is part of a framework that is also used on real robots. Since any real-world experiment also requires simulation, the things you will learn about the simulator itself will be valuable later on as well.

In chapter 2, we will start with the basis of reinforcement learning: balancing learning about the environment with exploiting that knowledge. This will introduce you to the action-value function Q, learning rate α, exploration rate ϵ, the ϵ-greedy and softmax exploration strategies, and different initialization strategies.

Chapter 3 continues with the concept of Markov Decision Processes and solving them using dynamic programming, specifically the policy iteration and value iteration techniques. You will learn about the state-value function V, delayed rewards, and the discount factor γ.

Chapter 4 introduces you to your first real reinforcement learning problem and the concept of temporal difference learning for solving it, using the tabular

SARSA method. In this chapter you will also encounter the `matode` interface to the Open Dynamics Engine for dynamics simulation, and the state and action discretization needed to use tabular SARSA on real-world problems.

To speed up the learning process, chapter 5 presents eligibility traces, which use the trace falloff rate λ. You will use the tabular SARSA(λ) method with replacing traces to speed up your dynamics simulation. This chapter also introduces you to the `XML` file format used by `matode`.

Chapter 2

Exploration and exploitation

Reinforcement learning is about learning by interaction with the environment. After each action taken, the learning agent gets a reward that tells it how good the action was. Importantly, because it is an *unsupervised* learning technique, the agent is not told what the best action was. It will therefore have to find out by itself whether a reward of, say, 100 is good or bad with respect to the other possible actions. To achieve this, it will need store the reward it gets for each action in an action-value function, $Q(a)$. A *greedy policy* will then simply choose the action with the highest stored reward. Two problems are that there might be too many actions to try exhaustively, and that the rewards may be *stochastic* such that the *average* values needs to be stored, which only converge slowly to the real *expected reward*. **book:** Ch. 2

The agent will have to find a balance between *exploiting* its knowledge of Q using the greedy policy, and *exploring* different actions to better estimate the real Q^*. We will investigate this exploration vs exploitation tradeoff using the n-armed bandit problem.

2.1 The n-Armed Bandit

The n-armed bandit is a slot machine with n arms, each of which has a normally distributed reward with a certain mean and variance 1. For a particular machine, the means are chosen according to a normal distribution with mean 0 and variance 1. The goal is to maximize the total cumulative reward over a number of plays. **book:** 2.1

Exercise 2.1 Look at the code you have been given for this chapter: `testbandit.m` and `getbandit.m`. Try to find out how a bandit is initialized and how it is run. Using the Matlab command line, create a 2-armed bandit for yourself and crank the arms a few times. Which arm do you think gives the greatest average reward? Does that correspond to the second output of the `getbandit` function?

The `testbandit` function provides the structure for this exercise. It creates `tasks` `N`-armed bandits and plays each one `plays` times. It records how many times each action is taken, the average reward, and the average number of optimal actions taken. The only thing it doesn't do is the actual learning and action selection.

book: 2.2

Exercise 2.2 Implement ϵ-greedy action selection in `testbandit`. The exploration rate is called `eps`.

book: 2.5

Exercise 2.3 Implement incremental sample averaging as the Q-update rule and run the test with $\epsilon = 0.1$. Plot the results and verify that they are the same as Figure 2.1 in the book.

ϵ-greedy is only one way of exploration; it explores all actions equally. A different approach is *softmax*, which chooses actions with high average rewards more than low-reward actions. There is no consensus about which one is better.

book: 2.3

Exercise 2.4 Implement softmax action selection using the Gibbs distribution. Use the following function to sample from a distribution:

```
function a = sample(p)
%SAMPLE Sample from a discrete distribution
%   A = SAMPLE(P) samples an action according to distribution P.
%   P is a vector of probabilities for each action. P does not
%   have to sum to 1, in which case it is normalized.
    pc = cumsum(p);
    a = find(pc > pc(end)*rand, 1);
end
```

Try different temperatures. Can you find a setting that works better than you ϵ-greedy run in Exercise 2.3? Compare the reward and optimal action plots between the two methods and explain the difference.

Up until now, we have been using incremental sample averaging as the learning mechanism. While this is good for theoretical results, it is not often used in practice. Often a fixed step size α is used, which is especially helpful on problems that change over time.

book: 2.6

Exercise 2.5 Modify your update rule to use a constant step size, α. Find a value of α that performs at least as well as sample averaging as far as the average reward is concerned. Use ϵ-greedy action selection.

When you look at the `testbandit` script, you can see that the Q-table is initialized to all zeroes. This introduces a bias when selecting an action, especially in the constant-α case. This bias can be used to encourage or discourage exploration. After all, if the initial values are higher than the real rewards, the agent will prefer to choose unexplored regions which still have those high Q-values.

Exercise 2.6 Initialize the Q-table to uniformly distributed random values between 0 and 1, rerun `testbandit` and compare the results to those in Exercise 2.5. Explain the results (they are most clear when looking at the optimal action plots). **book:** 2.7

Chapter 3

Dynamic programming

In the previous chapter we dealt with a stateless problem. In the real world, which action is best almost always depends on the *state* of the system, such as the position of a robot. Reinforcement learning deals with a special type of system in which the way a state is reached – the history of states and actions that precedes it – is irrelevant, i.e. the state itself summarizes past sensations compactly. These systems are said to have the *Markov property.* In robots, this often means that the state should include the joint positions as well as their velocities.

book:
Ch. 3 and 4

This chapter doesn't yet deal with the full reinforcement learning problem, but instead focuses on deciding optimal actions in Markov systems with known dynamics: the problem of solving *Markov Decision Processes* (MDPs). MDPs with unknown dynamics are the subject of the next chapter.

An important difference between a general MDP and the n-armed bandit is the concept of delayed rewards: while for the bandit the reward was always immediate, an MDP may perhaps only give a reward after many state transitions (time steps), for example when a robot hand has picked something up. The agent then has to propagate this information to earlier states and actions. It does this by not only keeping track of the expected immediate reward of an action, but also the cumulative reward of following the same policy afterwards. To avoid infinities, this *return* is discounted with a *discount factor* γ the further it lies in the future.

The return is again tracked through the value function Q, this time also indexed with the state s and policy π, as $Q^{\pi}(s,a)$. As we are dealing with known dynamics for now, we can calculate which action to take in order to get to a certain successor state. This means we don't need to store the expected return for each state-action pair: the return for each state suffices. The result is a *state-value function* $V^{\pi}(s)$.

We will now look at a set of methods for calculating and optimizing V, known as *dynamic programming* techniques.

3.1 The grid world

book: Example 3.8

Our toy problem for this chapter is a simple grid world. This is a 5x5 grid in which the agent can move according to the four cardinal directions. Movement to a square outside the grid is prohibited, instead resulting in a reward of -1. Positive rewards are awarded for reaching certain points A (+10) and B (+5), after which the agent is placed at specific successor points A' and B'.

The goal is to find the optimal action for each state. An optimal policy π^* will be *greedy* with respect to the optimal value function V^* , i.e. in each state s it will take the action that leads to the state s' with the highest $V^*(s')$. Our first solution method, called *policy iteration*, uses this property to successively approximate V^*. It interleaves *policy evaluation*, which calculates V^π, and *policy improvement*, which improves π by making it greedy with respect to V.

book: Figure 4.3

The policy iteration algorithm in the book is given for stochastic transition functions. The grid world uses deterministic transitions, which simplifies it somewhat. Most importantly, line 8 of Figure 4.3 can be rewritten as:

$$V(s) \leftarrow \mathcal{R}(s, \pi(s)) + \gamma V(\mathcal{T}(s, \pi(s)))$$

where $\mathcal{R} : S \times A \to \mathbb{R}$ is the reward r for taking action $a \in A$ in state $s \in S$, and $\mathcal{T} : S \times A \to S$ is the successor state s'. In the Matlab function `gridworld`, these have been combined in a single function `[`r`, `s'`] = observe(`s`, `a`)`. Similarly, line 15 can be rewritten as:

$$\pi(s) \leftarrow \underset{a}{\operatorname{argmax}} \left(\mathcal{R}(s, a) + \gamma V(\mathcal{T}(s, a))\right)$$

book: 4.1

Exercise 3.1 Implement the policy evaluation step in `gridworld.m`. Use the given θ of 10^{-5} as the stopping criterion.

book: 4.2

Exercise 3.2 Implement the policy improvement step, and run the algorithm. Verify that the results are the same as Figure 3.8 in the book (use `plotv`). How many observations did it take to converge?

The value of γ is a measure of how much into the future the algorithm looks. Lower values require fewer iterations to converge, but make it short-sighted.

book: 3.3

Exercise 3.3 Use a value of $\gamma = 0.8$. What changes in the resulting policy? How about the required number of observations?

Policy iteration has the disadvantage that it uses multiple iterations in the policy evaluation step, slowing down the convergence. It has been proven that is not necessary to wait for full convergence during policy evaluation. Instead, it is possible to alternate a predetermined number of policy evaluation sweeps with policy improvement steps. The special case of one policy evaluation sweep, called *value iteration* is particularly easy to implement, because the policy evaluation and policy improvement steps can be coalesced into a single update.

Exercise 3.4 Implement value iteration. Since `gridworld` outputs V instead of π, you can skip the last two lines of Figure 4.5 in the book. Verify that the results are the same as using policy evaluation. How many observations did it take this time? **book:** 4.4

It is not necessary to sweep through the state space deterministically. This will help us later on, because in many real-world cases it is not even possible to do this (imagine successively putting your robot in all possible configurations, including joint speeds!). While real RL uses actual real-world trajectories, for now we will do this asynchronous updating randomly.

Exercise 3.5 Implement asynchronous value iteration by modifying your value iteration loop to have a fixed number of 250 iterations, and choosing s randomly for each iteration (use `randi`). Verify that only 1000 observations are made and compare the resulting V with the V^* you found in the earlier exercises. Also compare the policies, and explain the results. **book:** 4.5

Chapter 4

Temporal-Difference Learning

It is at last time to work on a full blown reinforcement learning problem with multiple states, an unknown transition function, and which cannot be randomly initialized. To solve this problem, we use *temporal difference learning*, which is closely related to asynchronous value iteration as you implemented it in Exercise 3.5. The main difference is that the updates are sampled along the trajectory of an agent *while using π as a control policy.* Because they are samples (remember Exercise 2.1?) V is adjusted incrementally towards the new value using a learning rate, α.

book:
Ch. 6

This is also the first time we will be using a proper dynamics simulation representing a real system, which we'll introduce now.

4.1 Simulation

As backend simulator we will be using the Open Dynamics Engine (ODE) by Russell Smith. While ODE is slower than direct model simulation for most of the toy problems you will be working on, the advantage is that it can be directly applied to the more complex systems you will encounter in your future projects. From the ODE website at `http://www.ode.org/`:

> ODE is an open source, high performance library for simulating rigid body dynamics. It is fully featured, stable, mature and platform independent with an easy to use C/C++ API. It has advanced joint types and integrated collision detection with friction. ODE is useful for simulating vehicles, objects in virtual reality environments and virtual creatures. It is currently used in many computer games, 3D authoring tools and simulation tools.

An ODE model consists of *bodies* (which have a certain mass and inertia), *geometries* (defining the shape of a body for collisions) and *joints* (connecting the bodies). A *motor* can be defined on a joint to apply a certain force to it. Different joint types – such as hinges or sliders – are available, providing all the tools to model many different robots.

ODE works by fixed-timestep numerical integration of the equations of motion that define the system of connected bodies. When bodies collide, it creates temporary "joints" at the collision point that make sure the bodies do not intersect. These joints constitute a spring-damper system with finite K and D. Both the fixed step size (causing collision to be detected some time *after* they've taken place) and finite K/D values (meaning bodies can "sink into" one another a tiny amount) lead to inaccuracies. With well-chosen values for these parameters, ODE can be remarkably accurate.

4.1.1 `matode`

Our simulator, `matode`, is a Matlab interface to ODE. It uses Matlab's object oriented programming capabilities to make the interface as easy as possible. The main class is `odesim`, which provides global interaction with the simulator such as initialization, running a simulation step, and resetting to an initial condition. It also allows you to retrieve sensor and actuator indices, with which you can sense joint positions and drive motors. The the constitute the robot and the joints that connect them are defined in an `.xml` file provided by us and loaded during initialization.

Exercise 4.1 Install `matode` from the zipfile on Blackboard. Under Windows, adding the toolbox to your Matlab path is enough. Under Linux, you should make sure that `libmatode.so` is in your `LD_LIBRARY_PATH` as well.

A typical script involving `matode` looks as follows:

```
sim = odesim('mountaincar.xml');                % Load configuration
vel = sim.sensor('robot.base.velocity.y');      % Define sensor
motor = sim.actuator('robot.motorjoint1.torque');% Define actuator
actuators = sim.actuate();                      % Get actuation vector
for t = 0:sim.step():6                          % Simulation loop (6s)
  sensors = sim.sense();                        % Measure sensor values
  if sensors(vel) > 0                           % Read sensor
    actuators(motor) = 0.5;                     % Set actuator
  else
    actuators(motor) = -0.5;
  end
  sim.actuate(actuators);                       % Run simulation step
  pause(sim.step());                            % Run in real-time
end
sim.close()                                     % Destroy simulation
```

Note how the sensor has been defined: it gets the absolute `velocity` in the `y` direction of the `base` body in the `robot` object.

This simulation uses a simple controller to drive a car up a hill. While the car doesn't have enough torque to accomplish this task immediately, it can use another hill to gain speed. This is called the mountain car task.

Exercise 4.2 Find out, using `help odesim`, the different things you can use as a sensor. Copy the above script and modify the sensor definition such that it senses the velocity using the turning speed of the wheel axis. Run the script to verify that it works.

4.2 The mountain car task

To solve the mountain car task using reinforcement learning, we need a reward signal as well as a state definition that has the Markov property. For a reward, `mountaincar.xml` defines a `goal` that should be reached. The reward for coming within 0.2 meters of this goal is 100 (in which case the episode is terminated), while every time step carries a negative reward of -0.1.

Exercise 4.3 Find out how to use the distance between the `base` of the car and the `goal` as a sensor. The `goal` body is part of the `goal` object. Edit `mountaincar.m` and define the goal distance sensor, the reward calculation and the termination criterion.

To be Markov, the state of the mountain car has to include its position as well as its velocity. As is often the case in the real world, these values are continuous. To work with our tabular algorithms they need to be discretized. There are three possible actions: full negative torque, zero torque, and full positive torque.

Exercise 4.4 Define a sensor for the y position of the car, and implement the state discretization. You may use the following function:

```
function s = discretize(v, maxv, numstates)
%DISCRETIZE Discretize and clip a continuous value.
%   S = DISCRETIZE(V, MAXV, NUMSTATES) discretizes V,
%   from domain [-MAXV, MAXV] into a discrete state S in range
%   [1, NUMSTATES].
    x = (sign(v).*min(abs(v), maxv) + maxv)./(2.*maxv+0.001);
    s = fix(x .* numstates)+1;
end
```

Also implement the action calculation, which needs to map an action in the range `[1, actions]` to a torque in the range `[-maxtorque, maxtorque]`.

To actually use TD-learning, we need to go back to learning a state-action value function $Q(s,a)$, because the state transition probabilities (dynamics) of the environment are unknown and therefore π cannot simply be derived from V. The natural result is called the SARSA TD control algorithm.

book: 6.4

Exercise 4.5 `mountaincar.m` uses optimistic Q-value initialization to drive the exploration, so implement simple greedy action selection. Also implement the SARSA update rule; if you have used the suggested variable names, they should be the same as in the book (`sp` = s', `ap` = a'). Run your algorithm and plot the results using `plotq(Q, cr);`.

book: 3.4

Exercise 4.6 Inspect the Q table and compare the expected rewards recorded there to the actual rewards we defined in Exercise 4.3. What do you notice?

OPTIONAL

Try to modify your update rule to correct this.

END OPTIONAL

The time resolution and discretization of the state space can have a large effect on learning performance. Remember that the state of a reinforcement learning problem must have the Markov property. It expects that taking an action in a certain state always has the same effect (although the effect may be probabilistic). This property is often violated when we start to discretize an originally continuous system.

Exercise 4.7 Double the time resolution of the controller by editing `mountaincar.xml` and setting `steptime` to <u>`0.05`</u> and `subsamplingfactor` to <u>`20`</u>. What happens when you try to learn in this situation? (inspect the Q-table!). Try to correct by increasing the state space resolution (`states` in `mountaincar.m`) and number of episodes. Explain your findings.

Chapter 5

Eligibility traces

The plain SARSA algorithm you implemented in the previous chapter does not converge very quickly. The reason is that the expected cumulative reward value is only backed up once every time the path to the goal is taken; when the goal is reached for the first time, only the value of the direct predecessor state is adjusted! Eligibility traces overcome this problem by updating values based on the *eligibility*: how much influence they had on achieving the goal.

book: Ch. 7

First, however, you will model your own system in ODE, using the `matode` XML language.

5.1 Modeling

In the last chapter you used a mountain car simulation, defined in a file called `mountaincar.xml`. This file described all the bodies, joints and motors in the system, as well as collision information and settings for the integrator. We will now briefly explain the format of this file.

5.1.1 `constants`

The first important section is called `constants`. As the name implies, constants are defined here that can be used in the rest of the file. As with other numerical values, the constants may contain simple mathematical expressions. Constant expressions may also reference other constants, as long as those are defined earlier in the file (see Figure 5.1).

5.1.2 `ode`

The `ode` section defines the simulation environment, and takes up the rest of the file. It contains global settings, such as the `globalK` and `globalD` constants for the spring-damper physics simulation used by ODE, as well as the gravity

```
<configuration>
      <constants>
            <length>0.2</length>
            <radius>0.015</radius>
            <density>7874</density>
            <mass>length*_pi*radius^2*density</mass>
      </constants>
      ...
</configuration>
```

Figure 5.1: Using mathematical expressions in the `constants` section, referencing previously defined constants.

vector `gravityZ` (and possibly `gravityX` and `gravityY`, if you're so inclined). Most important for now are the duration of a simulation step, `steptime`, and how many substeps are done for each step, `subsamplingfactor`. In general, the step time depends on the bandwidth requirements of your controller, while the subsampling factor should be determined by the expected speeds and forces in the simulation.

The `ode` section also defines the objects in the simulation, and collision information (material properties, and which objects may collide). As the system that you will be modeling does not include collisions, we will focus on defining objects.

5.1.3 `object`

An object is a collection of rigid bodies connected by joints. An object has a `name` and zero or more `initialconditions`, which determine the body orientations when the simulation is reset. As such, the `initialcondition` defines the `bodyname` and a desired `rotation` of that body (see Figure 5.2). The object may also define `geom`etries associated with its bodies, which are used in collision detection.

Bodies represent the moving masses in an object. A `body` has a `name`, `mass` and a moment of inertia defined by `IXX`, `IYY` and `IZZ` (and possibly other combinations, depending on the shape of the body you're trying to simulate). It can also define how it should be drawn, using the `drawinfo` tag (see Figure 5.3). Note, though, that what is drawn may be completely unrelated to the moment of inertia, or even the collision geometry!

When drawing, the coordinate system is centered on the body. Any objects you draw (such as the cylinder in figure 5.3) are also placed in the center, unless you specifically move it with `x`, `y`, `z` position tags. The positive Z axis points upwards.

You can define an `anchor` on a body (and also in the plain `ode` section, in which case it is an anchor fixed in the world) in order to connect the body with

```
<object>
      ...
      <initialcondition>
            <bodyname>pole</bodyname>
            <rotation>
                  <axis>
                        <x>1</x>
                        <y>0</y>
                        <z>0</z>
                  </axis>
                  <angle>_pi</angle>
            </rotation>
      </initialcondition>
</object>
```

Figure 5.2: An initial condition for the `pole` body, rotating it 180 degrees around the X axis.

```
<body>
      ...
      <drawinfo>
            <cylinder comment="pole">
                  <radius>radius</radius>
                  <length>length</length>
            </cylinder>
      </drawinfo>
</body>
```

Figure 5.3: Using a cylinder as the graphical representation of a body. `radius` and `length` are constants.

others using `joints`. The anchor defines a point (`x`, `y`, `z`) on the body at which the joint is fixed (see Figure 5.4). The coordinate system is the same as that for drawing, so the anchor in the figure is located at the bottom of the body. Because you can have multiple anchors per body, it also has a `name`.

```
<body>
      ...
      <anchor>
            <name>world</name>
            <x>0</x>
            <y>0</y>
            <z>-length/2+radius</z>
      </anchor>
</body>
```

Figure 5.4: Defining an anchor.

A `joint` connects two anchors (`anchor1` and `anchor2`) and can be of different `types` (such as a hinge, slider, universal, etc.). Depending on the type it can have a number of properties, such as the axis along which the movement can occur (see Figure 5.5).

```
<joint>
      <name>joint</name>
      <type>hinge</type>
      <anchor1>
            <bodyname>world</bodyname>
            <anchorname>pole</anchorname>
      </anchor1>
      <anchor2>
            <bodyname>pole</bodyname>
            <anchorname>world</anchorname>
      </anchor2>
      <axisX>1</axisX>
      <axisY>0</axisY>
      <axisZ>0</axisZ>
      ...
</joint>
```

Figure 5.5: A hinge joint definition.

A joint can also have a `motor` associated with it, which can be actuated to provide a certain force or torque (see Figure 5.6).

```
<joint>
      ...
      <motor>
            <type>torque</type>
      </motor>
</joint>
```

Figure 5.6: A torque-controlled hinge joint motor.

Exercise 5.1 Look at `pendulum.xml` and try to understand the basic structure. Now look at `mountaincar.xml`, and give the names of all the objects and their constituent bodies.

5.2 Pendulum swing-up

The system we will be using in this exercise is an underactuated pendulum swing-up. The object is to swing up and balance a pole using a motor at one

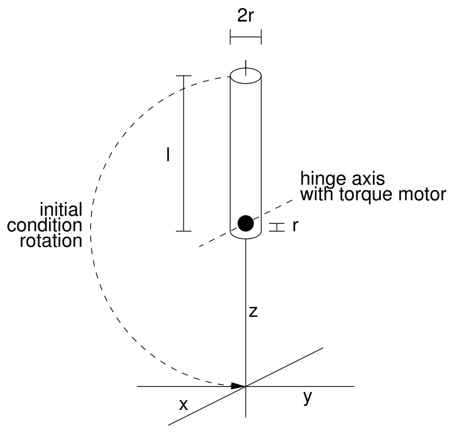

Figure 5.7: The pendulum model.

end. As in the mountain car task, the motor is not powerful enough to do this in one go, but has to swing back and forth before making the final swing-up.

Exercise 5.2 Model the pendulum swing-up in `pendulum.xml`. Use the comments in the file and the examples in section 5.1 to see what goes where. You can also look at `mountaincar.xml` for further guidance. Note that we want to have the zero position with the pole at the top; the easiest way of doing this is to connect the joint at the bottom of the pole and rotate it 180 degrees as the initial condition. Use an iron pole with $l = 0.2$ and $r = 0.015$.

In order to test your model, we will first construct a swing-up controller ourselves.

Exercise 5.3 Build a controller (see section 4.1.1 for the basic control loop) to swing up the pendulum, assuming the maximum torque `maxtorque` you can apply is 0.5. Use the following code to determine the actuation signal:

```
if abs(angle) < 0.5
    torque = -10*angle-0.5*anglerate;
else
    torque = sign(anglerate)*maxtorque;
end
```

You will have to define sensors for the `angle` and `anglerate` of the hinge joint, and an actuator to apply the `torque`.

If the controller you built managed to swing up the pole, it's time to create a learning algorithm for it. First, we'll start with the same SARSA algorithm you used in the previous chapter. Be sure to reuse your code, but note that there is no termination criterion this time!

Exercise 5.4 Create a SARSA algorithm to learn how to swing up and balance the pendulum. Use the angle and angular velocity of the hinge joint as the state vector; discretize these to 19 states each over the interval $[-\pi, \pi]$ rad and $[-20, 20]$ rad·s^{-1} respectively. Use ϵ-greedy action selection with $\epsilon = 0.05$, random Q initialization in the interval $[0, 1]$ and the other parameters as in `mountaincar.m`. Use the following equation as the reward function:

$$r = -5\phi^2 - 0.1\dot{\phi}^2 - \tau^2 \tag{5.1}$$

where ϕ and $\dot{\phi}$ are the angle and angular velocity of the pole, while τ is the applied torque. You can read the torque as a sensor that is simply called `robot.joint.torque`.

To get a baseline for comparing against the use of an eligibility trace, we need to get an average convergence speed.

Exercise 5.5 Run your algorithm, and plot the results (including the performance graph `cr`) using `plotq`. Has this run converged after 200 episodes? If so, to which value? Run the algorithm five times and take the average of `cr` to confirm this result.

Now it's time to implement an eligibility trace. In the simplest case, the eligibility of a state-action pair is dependent on how long ago it was visited. The implication is that more recent pairs should get more credit for achieving a reward than those in the past. The eligibility e of a state-action pair is $\lambda^{\Delta t}$, with Δt the number of timesteps since the pair was last visited and λ the *trace decay parameter*.

In the tabular case, the implementation is rather simple. We simply keep track of the eligibility of each state-action pair, multiplying it by $\gamma\lambda$ every time step and setting the eligibility of the current state to 1. To update Q, we multiply the learn rate α by the desired change in value δ and the eligibility e for each state-action pair.

book: 7.5

Exercise 5.6 Implement SARSA(λ) with a *replacing* eligibility trace in your algorithm (eq. (7.14) in the book, (7.17) on the web). Use $\lambda = 0.92$ and make sure to vectorize your Matlab code – it should not take more than a few lines. Now repeat Exercise 5.5 and compare the results.

book: 7.8

Exercise 5.7 Use regular accumulating traces (eq (7.10) in the book, (7.13) on the web) and compare the performance to ex 5.6. Why is there such a difference? Think about what happens when the pendulum is balancing at the top.