# Lab 5 TDDC17

Oskar Håstad, Martin Hinnerson

Version 0.1

2018-10-25

**Status**

| | | |
|---|---|---|
| Reviewed | | |
| Approved | | |

# Project identity

2018/Autumn term
Tekniska högskolan vid Linköpings universitet, IDA


| Name | Telephone | Email |
|---|---|---|
| Oskar Håstad | | oskha905@student.liu.se |
| Martin Hinnerson | 072 210 43 72 | marhi386@student.liu.se |

**Course responsible:** Patrick Doherty, patric.doherty@liu.se
**Lab Assistant:** Axel Strandberg, axest863@student.liu.se

# 1 Part 2

Here we implemented the Q-learning Angle Controller. The purpose of this part is to control the angle of the spaceship.

## 1.1 Question 1

(a) The states for the angle controller is dependent on the angle of the rocket. We will set a range of angles by defining `minAngle` and `maxAngle`. Outside of these angles we never want the spaceship to be and therefor the reward for being outside of these will always be zero. Now we also have to discretize the angle states so that we have a finite amount of states, otherwise the amount of states would be way to many and only limited by how many numbers the `double` datatype could represent.
After some testing our final parameters can be seen in tab. 1. We use only 4 possible actions, `no action`, `forward`, `left` and `right`. The reward function is made

| | |
|---|---|
| angleStates | $15$ |
| minAngle | $-\frac{\pi}{6}$ |
| maxAngle | $-\frac{\pi}{6}$ |

Table 1: Parameters for the Angle Controller

so that if the angle is outside of the maximum and minimum angles we return zero. However if we are inside of the allowed angles the rewards starts with 1 and then subtracts points for how far away from perfect it is. In the beginning we used the linear function seen in equation 1. Note that because we only want positive rewards, but this function can return negative numbers if the angle is bigger than 1. This is solved by simply limiting `maxAngle` and `minAngle` to 1 and -1 respectively.

$$reward = 1 - |angle| \tag{1}$$

This gave ok results but later, after we had implemented the full hover controller this was improved, see section 2.

(b) The Q-learning formula used can be seen in equation 2, which is the same as equation 20.8 in the course book.

$$Q[s,a] \Longleftarrow Q[s,a] + \alpha(N_{sa}[s,a])(r + \gamma * max_{a'}Q[s',a'] - Q[s,a]) \tag{2}$$

This function is called every time the agent moves. We can see that the Q-value is dependent on the previous state, action and reward. The maximum Q-value is multiplied with the discount factor, $\gamma$. The discount factor can be seen as a factor of how greedy the agent will be and thus this multiplication will represent the future. A smaller $\gamma$ will therefore result in an agent that will prioritize immediate reward, in other words a more greedy agent.
`r` in equation 2 is the reward explained above. This is used to drive the agent towards good behaviour.

The Q-value is also dependent on the learning rate $\alpha$. The learning rate will scale how much our earlier Q-values affect the new one. In the equation used this is also dependent on how many times we have tested an action in the current state.
So to sum this up, a Q-value is basically a value that tells how good a certain action is in a certain state. The hash-table in our agent will be updated with better and better Q-values for the good action in the different states when the agent is learning. After sufficient training the agent can then use this table to know what action it should perform in the state it's currently in.

### 1.2 Question 2

If exploration is turned of from the beginning the agent will not visit as many states when learning. This will result in high Q-values for a smaller number of states. If we never explore all the states the agent might still learn to fly fine but the solution will most likely be sub optimal. What could happen is that the agent could converge towards a Q-value it thinks are the best one but it could have found a better one if it would have explored more options.

## 2 Part 3

For the full hover controller we also added states and reward functions for the velocities in x and y direction. The state parameters used can be seen in tab. 2. In the beginning

| | |
|---|---|
| velocityStatesX | 5 |
| velocityStatesX | 10 |
| minVelocityX | -0.5 |
| minVelocityY | -0.5 |
| maxVelocityX | 0.5 |
| maxVelocityY | 0.5 |

Table 2: Parameters for the Hover Controller

the reward functions for the velocities were implemented similar to the one used for the angle controller. With these linear reward functions the agent was able to fly, but even after a long time learning there was still significant drift in the x and y directions. We wanted a bigger drop off in reward when the agent was not in the optimal states. First we tried an exponential function but the result was similar and after some thinking we realized this didn't do what we wanted. Instead a inverse exponential function was implemented. The final equation used was equation 3 and we used it for vx, vy and the angle reward.

$$reward = 1 - scaleFactor * sqrt(|param|) \tag{3}$$

The constant `scaleFactor` was used to scale the function for optimal behaviour, however the final solution uses `scaleFactor = 1`.

The reasoning behind using the square root is explained in fig.1. We wanted a faster decrease in reward and as seen in the graph that is achieved with the new function. Figure

1 also clearly shows that the absolute maximum value of the parameters is 1, otherwise the function can return negative values. The total reward is then simply the addition of

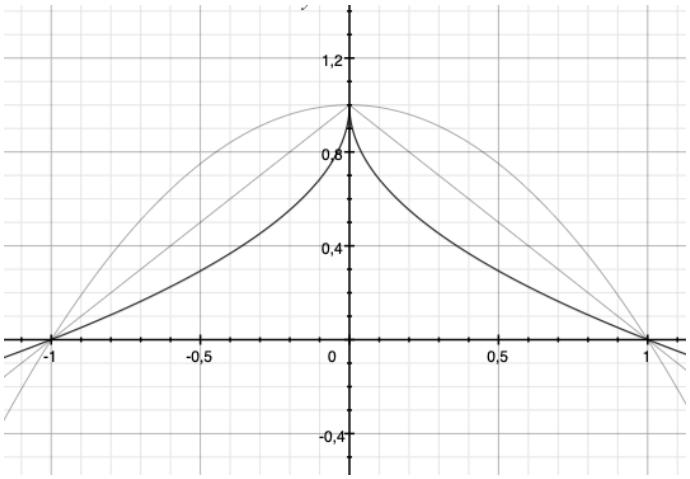


Figure 1: Reward Functions, the highlighted function corresponds to equation 3

the 3 different reward functions for `angle`, `vx`, `vy`.

With this implementation the rocket can fly with almost no drift, however it takes some training. If we let it train for more than a million iterations there is most of the time almost no drift and the rocket can parry pretty well if we give it manual input. By using the `scaleFactor` the function could have been made even steeper to punish drift even more. We did some testing with this but didn't see immediate improvement so the `scaleFactor` was left at 1.

## 2.1 Question 3

(a) As we see in figure 1 our reward function is not linear. However you could probably get an ok result with a linear model. To be able to find a good parameter w1 we would need some type of performance measure for the function. Here we would have to use a loss function and try to minimize it.

(b) Another model that could be used is a Artificial Neural Network. Here every neuron use a linear regression model of all the inputs passed through a non/linear activation function, traditionally a sigmoid. Each neuron get its inputs from all neuron in the previous layer. These different layers allows the network to better capture different layers of abstraction.

## 2.2 Question 4

For this scenario deep learning is our best option. A Convolutional Neural Network (CNN) could be used to feed the raw pixels as states to the Q-function. Different pooling layers could be used to downsample the information. This is essentially what they used in the ATARI video-game example from the lecture.