Stage 2 Report
Pushing a Box in a Straight Line

Jose Luis
Claus Meschede
Jan Rudolph

July 10, 2016

1 Robot control in the simulator

1.1 Movement implementation and odometry

According to our timeline, implementing the basic movements turn (for a specified angle) and go straight (for a specified distance) inside the simulator was the first and as we thought, least involved task. Unfortunately, there were a surprising number of obstacles to tackle, so this takes more time than we initially thought. First of all, the python code that enables us to control the ePuck in the V-REP simulator with a similar API as the vendor-supplied bluetooth remote serial console python module, was missing some pieces that were essential for our problem. There was no easy support for retrieving and changing the position and orientation of objects inside the simulator, so we went ahead and implemented an API to manipulate the V-REP object tree without having to deal with C function wrappers and number return codes. The second thing that was missing was support for the rotary encoder sensors of the e-Puck in the simulator, that represent the only way to get feedback from the wheels on the real robot and are thus essential to our movement problem.

Implementing the movement functions themselves wasn't straightforward either. There is no support in the API to make the wheels turn a specific angle or number of steps, one can only control the speed of the motors. Inside the simulator, the impulse is calculated so that quick changes in the motor speed lead to slipping of the wheels and an enormous impulse for the ePuck that makes boxes fly from the simulation area. The solution there was a linear acceleration and deceleration and a capping of the maximum speed.

It also seems like the speed of the wheels cannot be set at the exact same time, which makes going straight very hard. There is a mixture of synchronous and asynchronous api calls for the simulator, and we were not able so far to control the simulation steps synchronously in order to have a defined point for setting parameters.

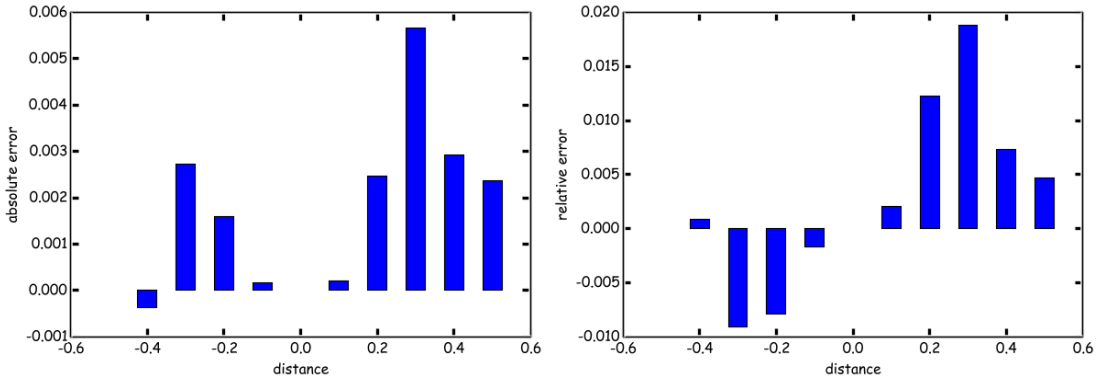

Figure 1: Plot of absolute and relative error when running the test movements going straight using the position API of V-REP, using the ODE engine. Not that this error is not a cumulative one.

The simulator itself also makes this very hard: When using the bullet engine, the ePuck robot moves and turns around even when only starting the simulation. We tried locking the joints inside the simulator, a feature created in order to simulate motor brakes, but the problems seems to be lying in the collision calculation, as this did not lead to any improvements. Switching to the Newton simulation engine helped with the random movements, however the engine is still experimental and experiences a number of glitches. The ODE engine also did not display major stand-still movement.

Apart from moving the robot by setting the joint speeds, we also implemented moving the robot by setting new coordinates using the V-REP remote api. Of course, this movement has to be done in very small incremental steps so that the collision system and the impulse calculations don't make the simulation completely unreasonable. Additionally, when moving the ePuck by setting it onto a new position or orientation, the wheels of the robot are not attached to the body, and have to be moved into the new position or orientation manually.

In order to validate the precision of the movements in the simulator, we wrote test cases that move the robot forward and backward for increasing distances and keep track of the error in the position. The results so far are very discouraging, as even with very small step-sizes and by using the cheating, e.g. using the V-REP position API the errors can be as high as a couple of percent, and the cumulative error eventually makes the robot flip over or move his point of origin and direction so far off that it falls of the simulation ground. This happens for both the Newton and the default engine of the simulator. As a result, before we continue this undertaking, we want to look at the real robot to verify whether it is as bad there as it is in the simulator.

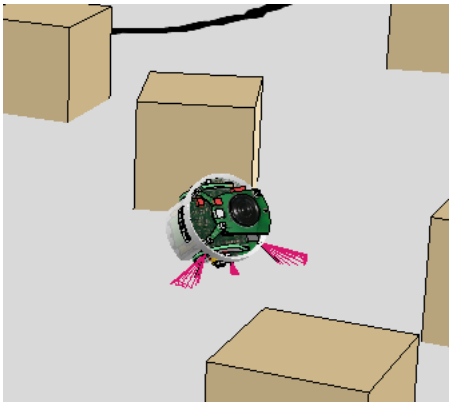

Figure 2: Moving in a straight line in a simulator is hard. The screenshot shows a flipped ePuck during a movement test run.

1.2 Box center tracking

In this project, the box center tracking will be done by using the proximity sensors of the ePuck robot. We used the simulation to place a box at different known angles around the robot and recorded the sensor output. Using this data, a simple nearest neighbor regression algorithm computes an angle for a proximity value input.

1.2.1 DOFs

When sensing a box in front of the circular ePuck robot with the proximity sensors, the proximity values will give the distance from the sensor to a tangent touching the robot. The side of the box facing the robot lies on this line. Unfortunately, it is not possible to say where on this line the box is exactly by using the proximity sensors alone. Therefore, in our problem, we want to physically restrict the box to stay in the middle of this tangent and limit the degrees of freedom for the box while pushing.

1.2.2 Simulation shortcomings

There are some issues in the simulation regarding the physics of the pushing problem. First of all, it is difficult to simulate the exact physical behavior of the box in the simulation, especially if we have to restrict the degrees of freedom. Furthermore, the bounding box of the ePuck robot and the position of the proximity sensors do not correspond very well. This leads to overlapping of the sensor positions and the box position causing incorrect sensor measurements.

1.2.3 Tracking in the simulation

To overcome these shortcomings of the simulation, we used the information given by the simulator to compute the box center position with respect to the robot. Of course,

this approach is only sufficient for implementing and trying the reinforcement learning algorithm in the simulation and it can't be transferred to the real robot.

2 Reinforcement Learning

2.1 Implementation

For solving the learning task we implemented a Q-learning algorithm in a small state space. The algorithm can choose from 3 different actions corresponding to straight movements in slightly different directions towards to box. After performing a chosen action with a predefined stepsize, the robot computes its new state by taking the new box position into account. The different states correspond to the direction in which the robot and the box are aligned. A reward is then given when reaching a certain state (e.g. state 0).
Actions are chosen with an eps-greedy policy. After reaching the reward state, the robot performs one further action and is then reset to a random initial state. Epsilon is defined as e^{-g}, with e being the number of episodes and g being a factor > 0 to control the greediness of the algorithm.

2.2 Testing

The implemented algorithm is able to learn the desired behavior after ca. 20 episodes. In the Q-Learning update step we set the learning rate α to 0.5 and the discount factor γ to 0.8. The greedy factor g regulating the tradeoff between exploration and exploitation has been set to 0.4. Figure 3 shows the Q-values learned by the algorithm. In the rewarded state 0, it has learned to aim at the boxcenter, in the other states, it has learned to choose the direction that leads to the rewarded state 0 as fast as possible.

State	Left	Straight	Right
0	109.69	**178.71**	123.05
1	17.82	106.81	**223.93**
2	0.	0.82	**156.94**
3	13.21	2.21	**120.15**
4	5.27	3.75	**62.23**
5	**98.82**	88.40	19.28
6	**194.16**	39.75	36.16
7	**148.56**	131.15	116.10

Figure 3: Q values after Learning in the simulator