RESEARCH PROJECT
FINAL REPORT

Student: Ionut Vintu
University of Stuttgart
Faculty of Computer Science, Electrical Engineering and Information Technology
Study programme: M.Sc. Computer Science

Coordinator: Prof. Dr. rer. nat. Marc Toussaint
1 Motivation

When applying reinforcement learning algorithms in the context of simulation, the experiments one conducts are only limited by the processing power of the computer at hand and by the available time. When it comes to using the same approaches on the real system, the strategies derived from theory one would like to use are not an easy, perfect fit anymore.

In the context of simulation there are no real sensors providing measurements, so the agent learns based on fully simulated data. Even if noise is added to it in order to mimic the reality of sensor data, it will not fully reproduce this concept. The sensor data is prone to errors and noise, which will be propagated into the learning process, affecting it in a more or less negative way. The other goal of this research project is to make use of a more reliable data source in the learning process, by introducing OptiTrack motion capture system in the working scenario.

2 Subsystems

How does the whole system work? Let’s break it down so every component is better understood. The whole system is made of subsytems, as presented in Figure 1. Racer plays the part of the agent, OptiTrack streams data regarding Racer’s pose, the experimental controller monitors it and collects the learning data and ZeroMQ provides the communication between the agent and its outer controller.

![Figure 1: Representation of the data flow among the subsystems of our working scenario.](image)

*OptiTrack streams data to a publisher node, which publishes it to tf topic. The experimental controller subscribes to this topic in order to have access to this data. The experimental controller and Racer communicate via ZeroMQ messages, in a bidirectional way. A more detailed presentation regarding this data flow will be delivered in the following sections.
In the following sections each subsystem is briefly described. The focus of this research project is set on Racer and OptiTrack motion capture system. The way they work and where used within this research project is also presented. At the end of their corresponding section, some of the encountered challenges are mentioned. These can be considered as open questions and matters that can be improved.

2.1 Racer

2.1.1 Short presentation

Racer is a wheeled inverse pendulum, which moves in $xz$ plane. More details about its components (battery, motors, encoders etc.) and how its dynamics is modeled can be found in mlr_students git repository → racerRobust branch → mlr/share/projects/racer/wiki. From now on mlr_students git repository → racerRobust branch → mlr/share/projects/racer will be referred to as path.

Several threads are running in parallel on Racer. For example, in path/racerRobust/main.cpp, which is the code that runs on Racer in the context of this research project (referred to in the following as Racer’s code), RoboClawModule deals with the motors, encoders and battery. IMU_poller reads the IMU data, i.e. gyroscope and accelerometer data. KalmanFilter thread computes Racer’s estimated state based on this data. With the help of GamepadInterface, Racer can be controlled via the gamepad. The last used thread is ZeroMQ, which deals with the communication between Racer and the experimental controller.

These threads share variables, having read and write access to them. The access has to be explicitly stated when the threads are declared, using instances of ACCESS class.

More information regarding the use of threads is available in the comments within mlr_students git repository → racerRobust branch → mlr/share/src/Core/thread.h and thread.cpp and in the last part of the guide - path/wiki/mlr.pdf.

For testing matters, the code in path/RoboClawModule, path/testMotor, path/IMU-module, path/gamepad can be used to check each thread separately.

2.1.2 How it was exploited

The main() function of Racer’s code, accessible at path/racerRobust, consists of an infinite for loop, in which Racer can either stabilize or perform an action sampled from a parametrized policy (exploration). The stabilization is achieved via a PD control law, whose gains are read from MT.cfg file. In the code part corresponding to the exploration, the current state, action, encoder information and time are appended to a variable which will be sent to the experimental controller by ZeroMQ thread. The for loop breaks at Ctrl – C or when specific buttons on the gamepad are pressed.

The experimental controller tells Racer whether to stabilize or start exploring (in case it is stable). The decision is made based on Racer’s angle (angle between the vertical axis $z$ and pendulum’s axis) and angular velocity. The corresponding raw data is received from OptiTrack. More details about this matter will be conveyed in Sections 2.2 and 2.3.
When Racer starts and stops running, messages embedding this information are sent to the experimental controller. Racer does not stop running immediately after Ctrl – C or specific buttons on the gamepad are pressed. It waits until the stop message is sent to the experimental controller.

There are also two helpful flags in the code that can be used to decide the mode in which Racer runs, namely flagControl and flagStop. By default, flagControl is set to true, which means that the motors receive commands, through the controls variable. The flag can be switched to false if the second command line argument is noctrl. This no control mode could be used for testing, e.g. checking the values of some variables or debugging purposes, without worrying about the motors turning. Regarding flagStop, it is by default set to false, which is equivalent to running the for loop until Ctrl – C or specific buttons on the gamepad are pressed. The flag can be changed to true by writing stop after ./x.exe in the command line. Through this switch, the motors receive 0 value command, i.e. they stop turning and the for loop immediately breaks. This behaviour mode is helpful when the program crashes and the RoboClawModule thread did not close properly, so the motors are still turning. Instead of cutting the power supply, the code could be run with stop.

Regarding the threads running on Racer, their looping mode, i.e. how they iterate step() function, is specified in their constructor. The following threads, RoboClawModule, IMU_poller and ZeroMQ loop with full speed and no idle time. Because KalmanFilter listens to imuData variable, it calls step() only when this variable was written into by IMU_poller. The GamepadInterface calls its step() every 0.01 seconds. The for loop in main() first waits until stateEstimate variable is written into by KalmanFilter thread and then it writes into controls variable, accessed also by RoboClawModule and into learningData variable, accessed by ZeroMQ. Therefore, the frequency for writing into controls and into learningData is given by IMU_poller thread. We report a frequency of 50Hz based on our tests. Moreover, this means that the frequency of sending learningData messages to the experimental controller is also 50Hz.

2.1.3 Challenges

To the best of our knowledge, all previous code concerning Racer was written using an older version of mlr repository. Racer’s code was written based on the current one. Finding equivalent functions and figuring out what to keep or change in the code was quite a challenging task, mainly because the existing documentation is sparse. Some of the older examples, which can be found in mlr_stuff repository → racer branch → mlr/share/projects/racer or racer_learning, were revived according to the newest repository version.

Initially, one of the experimental controller’s tasks was to constantly monitor Racer’s battery level, so it can stop Racer from running when the level drops below a certain threshold. This task was taken into consideration because it was assumed that the battery level influences how Racer performs, in terms of driving. This continuous supervision could be accomplished by reading the battery level in the step() function of RoboClawModule and sending it to the experimental controller. This idea was tried out, but it failed. The code crashed and the error we got was "terminate called after throwing an instance of ‘USBSerial::Exception’ what(): Timeout reached (in USBSerial::Read) Aborted". We assumed that there is a conflict between reading battery level and sending duty cycles to motors/reading encoders from the same address (but not at the same time). In the current implementation, the voltage is read just once, in open() of RoboClawModule
and compared to a predefined threshold. For safety reasons, the lowest level the battery should reach is 10.5V. For a 5 minute experiment, the battery is drained around 0.2V. Taking into account these aspects, we chose 10.7V to be our threshold.

The control law applied to stabilize Racer (the reference angle is 0) is of PD type. Because the integral component misses, Racer’s angle does not converge to the reference perfectly (i.e. error between reference value and system’s angle does not go to 0). For the PD law case, the error oscillates around 0 value. It may happen that these oscillations become quite big, which leads to a very slow or no damping at all for them. The derivative component may destabilize the control loop and in such cases one finds himself in the situation of stopping Racer, so this oscillatory behaviour comes to an end. While running experiments, we had to stop Racer many times because of this issue. This problem delays the whole learning data acquisition process and in some extreme cases causes the falling of Racer. We also tried to move Racer a bit with the help of the gamepad, in order to somehow cancel the oscillations. Furthermore, we observed that Racer falls more often when the battery level was low. We believe that this happens because there is not enough power to reach the command computed by the PD law to stabilize Racer.

Another issue worth mentioning, which needs more closer inspection, is when Racer suddenly falls, without any error occurring. Short time after falling, the motors start turning again, at a high speed, to compensate for Racer’s falling pose (Racer is in the stabilization phase, i.e. PD law code). The first assumption was that one of the threads running on Racer stops looping (i.e. calling its \texttt{step()} function). To test if this is the case, every time a thread loops, it writes some data into its corresponding file. When Racer falls all of a sudden, these files were analyzed, in order to see if there is a relevant difference among the number of lines each file has. Table 1 shows some of the differences we have observed during experiments of varying running time. For a correct comparison, we also looked in the files after an experiment with normal behaviour, i.e. Racer did not fall. The corresponding values are displayed in the last column of Table 1 and are the equivalent of a baseline. Initially, we could not draw any conclusion with regard to this problem, because there was not a significant difference between falling cases and the baseline. After a more meticulous look, the files exposed the root of the falling problem. In the file corresponding to \texttt{IMU\_poller} there was a bigger time step between two written lines. This translates in \texttt{IMU\_poller} briefly stopping from looping. Following the threads’ dependency presented in Section 2.1.2, this matter automatically leads to the \texttt{controls} variable not being written into and also to the motors not receiving commands for a short period of time. Taking into account that the motors need to be given a command constantly, this pause causes them to stop turning.

The fact that the wheels mounted on Racer are not fixed properly and wobble while moving, combined with the uneven surface of the floor in the laboratory, makes Racer even more unstable and introduces some trembling in Racer’s behaviour. It is unknown how much this matter influences the exploration, stabilization and sensor data reading.
<table>
<thead>
<tr>
<th>Thread</th>
<th>No. of lines in file</th>
</tr>
</thead>
<tbody>
<tr>
<td>RoboClawModule</td>
<td>1151 16098 4768 8330 8310</td>
</tr>
<tr>
<td>IMU_Poller</td>
<td>1154 16102 4772 8342 8313</td>
</tr>
<tr>
<td>KalmanFilter</td>
<td>1154 16101 4772 8333 8313</td>
</tr>
</tbody>
</table>

Table 1: How many times a thread writes in the corresponding file, i.e. it loops. The values in the last column were obtained during an experiment when Racer did not fall, which represents the baseline for the falling cases.

2.2 OptiTrack

2.2.1 Short presentation

OptiTrack is a motion capture system, providing high-speed tracking via its cameras. A clearer and better documented presentation of OptiTrack is available at mlr_students repository → optitrack branch mlr/usr/VI.

In short, OptiTrack consists of cameras, markers and a motion capture software, called Motive. After the cameras are arranged such that the volume they capture fits the task at hand and the markers are placed as desired, the system needs to be calibrated.

Tracking data streamed by Motive is handled by mocap_optitrack ROS node. It publishes the data in tf topic, where it can be easily accessed by other nodes within ROS environment. More information about how to set up and properly use OptiTrack and mocap_optitrack can be found following the above mentioned path.

2.2.2 How it was exploited

In our working scenario, the cameras are placed such that their overlapping field of view contains the space where Racer mostly moves. The volume defined by this field of view can be visualized in OptiTrack software, as shown in Figure 2. The corresponding perimeter is roughly marked also outside the software by black tape stuck on the floor.

Initially, seven markers were set on Racer, but we observed that very often some are not visible during experiments. Now, ten markers are evenly placed on Racer and we believe OptiTrack can track it more robustly.

In order to track Racer’s pose, an OptiTrack body needs to be created from these markers. This creation moment is noteworthy, because the body’s orientation at that time represents its zero orientation set point, relative to OptiTrack coordinate frame. OptiTrack reference frame is given by the position and orientation of the calibration square in the real world, when setting the ground plane (which takes place in the calibration stage), as shown in Figure 3.

During experiments it may happen that Racer gets outside the viewed volume, so it needs to be brought back inside with the gamepad. Otherwise, the experiment should be ended because of unreliable data provided by OptiTrack. It should be noted that in our implementation, Racer can be controlled with the gamepad only in the stabilization phase.
2.2.3 Challenges

The working place is far from being ideal. The problem of the shaking floor was already mentioned in Section 2.1.3 which introduces unsteady and undesirable movements in Racer’s behaviour (its motion is not smooth and straight, but with jumps). The same floor, if anyone steps on its moving tiles, makes the cameras move. Therefore, the camera setup is changed and it needs recalibration, in order to preserve the reliability of OptiTrack data. Moreover, the camera setup can be influenced even if one steps on moving tiles outside the working area, because the shaking is further transmitted. As one could imagine, there are a lot of people moving in the laboratory and also around our working place. This situation forces us to do a calibration every working day. This challenge could be overcome if the cameras would be mounted on a more stable structure.

If the initial markers’ placement changes for some reasons (e.g. they get detached when Racer falls or weak adhesive), a new OptiTrack body should be created with the new configuration.

2.3 Experimental Controller

2.3.1 Short presentation

In our application, the experimental controller plays the part of an outer supervisor. Having access to OptiTrack data, it can monitor Racer’s pose, checking if it is stable or not, based on some criteria. Using ZeroMQ functions, it communicates with Racer. It handles the receiving and storing of learning data, by saving it into a file. If the reinforcement learning experiment requires noise to be added to the policy, the experimental controller also takes care of this matter. It basically controls the conducted experiment, thus its name was derived.

The experimental controller runs on a ROS node (for simplicity and hopefully, without loss
of understanding, we will refer to this node also as experimental controller). In Section 2.2.1 we
have briefly described how the flow of OptiTrack data is managed within ROS environment. As a
short reminder, mocap_optitrack node publishes the incoming data in tf topic. The experimental
controller is a subscriber to this topic, thus having access to OptiTrack data. Messages between pub-
lisher and subscriber node are passed to a callback function, in our code called optitrackCallback.
The calls to this function are stopped when Ctrl − C is pressed. The frequency of calling the call-
back function is given by the publishing frequency, which is 120Hz.

2.4 ZeroMQ

2.4.1 Short presentation

In broad lines, ZeroMQ is a messaging library, which allows the transfer of messages through
different types of sockets. Each kind of socket involves a specific communication scheme and
requires a certain messaging pattern. The basic socket patterns one could choose from are: request-
reply, publish-subscribe, pair, push-pull.

In order to get ZeroMQ running, you first need to install it and the easiest way to do that is by
following the installation instructions from the official website, [1] and [2].

There are also some examples provided at [3] and [4], which show how different sockets work,
considering a simple, straight-forward communication setup. Those interested in getting a better
sense of ZeroMQ’s functionalities should definitely go through these examples. For an even more thorough and complete understanding, one should read the dedicated guide, available online [5].

3 Determining the stability margins

This section describes a methodical way to determine the stability thresholds. To make a decision with respect to Racer’s stability, both pitch angle and its derivative (angular velocity, noted as ang_vel) are taken into account. In this context, two thresholds should be used and therefore, computed. For simplicity, we will note the stability margin for pitch angle with sm_ang and the one corresponding to its derivative with sm_vel.

Racer is unstable if the absolute value of pitch is beyond sm_ang or the absolute value of ang_vel is greater than sm_vel. These thresholds vary during the testing experiment, so the stability zone cannot be defined based on them, because it will implicitly vary. The stability region should remain the same for each tried out case, thus it is specified based on predefined, fixed, small thresholds (0.05 for pitch and 0.15 for ang_vel), to ensure a robust stable state for Racer when starting an episode.

Our approach consists of trying out systematically multiple combinations of values for these thresholds. We start from some initial values, sm_ang = 0.1rad (= 5.73°) and sm_vel = 0.2rad/sec. The increment step for both is 0.05. One could use a smaller value in order to obtain a more refined computation. The upper limit for both thresholds is 0.5. For each combination the goal is that Racer runs continuously, without falling. We impose each combination of values to be stability thresholds in the code in order to test if the PD law succeeds in preventing Racer from falling. For each combination, five episodes are performed. The number of episodes in which Racer falls is reported.

We ran three experiments, in which the command given to the motors is 0, 1000 and -1000, respectively. By conducting the last two experiments, we wanted to see if the stability margins are asymmetrical, i.e. different thresholds for positive and negative values, respectively. The results obtained during these experiments can be found in path/ExpController/stability_margin_results.

Unfortunately, we can not draw any solid, clear and well founded conclusions. We strongly believe that the acquired results were heavily influenced by a series of factors, such as battery level, PD law oscillations, uneven floor surface, Racer’s orientation when stateStable variable becomes true and Racer starts a new episode etc. The results are inconclusive in the sense that for some small values of the two thresholds Racer falls a couple of times and for larger values, it does not fall at all. For example, in the case of motor’s command equal to 1000, Racer falls 3 out of 5 episodes for sm_ang = 0.15 and sm_vel = 0.25, but it does not fall for sm_ang = 0.2 and sm_vel = 0.25 or sm_ang = 0.3 and sm_vel = 0.3. The same thing happens also for command equal to -1000. Racer falls 3 out of 5 episodes for sm_ang = 0.15 and sm_vel = 0.2, but it does not fall at all for sm_ang = 0.3 and sm_vel = 0.2.
4 Conclusion

During this research project we managed to build a framework in which an experimental controller supervises the entire learning process. It constantly monitors Racer’s pose, based on reliable data, provided by OptiTrack motion capture system. It checks whether Racer is in the stability region or not and sends it corresponding commands. If Racer is stable, it starts exploring, by sampling actions from the current policy. If it is unstable, it needs to stabilize, with the help of a PD control law. The communication between Racer and the experimental controller is ensured by ZeroMQ messaging library. The experimental controller’s main duty is basically to prevent Racer from falling, which leads to a continuous learning process. It also handles the learning data acquisition and processing and the policy updates, maintaining a constant execution flow.

The proposed framework successfully manages to prevent Racer from falling, but with significant drawbacks. We believe that the exploration phase is limited by this framework. The learning is not done efficiently, because Racer is not allowed by the experimental controller to fully experience the results of the performed actions. Therefore, how much it learns is constrained by the chosen stability margins. On one hand, small thresholds guarantee that Racer will not fall, but suppress its learning ability. On the other hand, larger values for the stability margins are not enough to keep Racer from falling, although they may be more permissive in terms of how much Racer learns. Instead of accepting this compromise, we suggest another solution that could be implemented at hardware level. Our recommendation refers to a hardware addition that prevents Racer from falling, while still allowing it to explore freely. Thus, Racer could reach orientations that are forbidden, i.e. considered unstable, within our framework. Figure 4 illustrates a possible prototype for the proposed hardware solution.

![Figure 4: A possible implementation of the proposed hardware solution, which could prevent Racer from falling and also let it explore other states.](image)

The method we developed to determine the stability thresholds for both pitch angle and angular velocity clearly showed that how Racer performs is highly influenced by factors such as battery level, uneven floor surface, PD oscillations, wobbling wheels etc. The effects introduced by these factors should be fully removed or at least reduced as much as possible, in order to obtain more promising results. We also believe that implementing a better control law on Racer to substitute the existing PD law will be able to stabilize Racer from a wider range of states.
References


