ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Universität Zürich[UZH]

Velocity Control and SLAM With Spiking Neural Networks for an Omni-Directional Robotic Vehicle

Report Semester Project

Mario Blatter
D-ITET

July 21, 2017

Advisors: Dr. Yulia Sandamirskaya, Prof. Tobi Delbrück
Institute of Neuroinformatics, ETH Zürich, Uni Zürich

Abstract

The project at hand consisted of two major components. Both involved an omnidirectional robotic vehicle (Omnirobot) with three Swedish wheels actuated by servomotors and equipped with a number of sensors (a ring bumper, an inertial measurement unit and and wheel encoders). Task one was the improvement of a software environment to communicate with the robots firmware in order to be able to navigate it and communicate with it in various ways. Furthermore, path integration was performed. The resulting data was used to visualize the robot's trajectory. Then, a velocity controller was implemented.
In the second part it was aimed at implementing an algorithm similar to SLAM (simultaneous localization and mapping) making use of the functions programmed in the previous part. The reconfigurable on-line learning spiking (ROLLS) neuromorphic processor was intended to be used for learning a simple map of the robot's environment using the device's bumpers as feedback and the plastic synapses on the chip. The entire software was run on the miniature computer board parallella, which connected all the hardware components.
However, due to the breaking of the robot's microprocessor board, only a simplified version of the second part could be performed. Namely, this was to show learning between neurons representing a simulated location and a 'collision population' as a proof of concept.

1 Introduction

Autonomous robotic navigation currently is a large field of research. Many application can be thought of being implemented towards goals such as optimizing trajectories, minimizing the error in movement, obstacle avoidance or simultaneous localization and mapping.
The latter is a common computational problem in robotic mapping and autonomous driving and has been studied extensively. It is know under the abbreviation SLAM and is an unsupervised machine learning problem. It deals with the fact that when trying to create a map of an environment and concurrently locate a robotic agent's position in it, errors inherent to the system will lead to a misinterpretation. One way to lessen this problem is the use of a controller. In this project, a standard PID controller will be implemented for this purpose. Additionally however, some sort of active feedback is needed, in order to continuously update and correct the map. This can be understood as a form of learning.
A simplified version of this mapping problem is aimed at in the second part of this project. But unlike most SLAM solutions learning will not be conducted in software but on a reconfigurable on-line learning spiking neuromorphic chip. The necessary elements that allow such an implementation are presented in the following paragraphs.

Omni-Directional Movement The idea behind omni-directional, or holonomic, movement is the facilitation of control and increased speed [1]. With the robot having three wheels positioned at an angle of 120° relative to each other, one obtains three degrees of freedom (the speed in x- and y-direction, plus the rotation around its own axis). This allows changing directions and orientation without having to do complicated maneuvres to correctly position the wheels as would be the case with two-wheeled robots.
There has been extensive work on how to control these types of robots by employing the corresponding kinematic models [1–4]. In this project, an interface for navigating an omni-directional robotic vehicle will be presented.

Localization: Path Integration and Visualization An important aspect for robot navigation is knowing its position at any time. Since relying on outside sensory information is not desired, self-localization is used. This is usually done by path integration, also known as odometry. Path integration is a means of recording and updating a vector to a given origin when moving. Two pieces of information need to be retrieved, the distance covered and the direction of movement [5]. Both can be obtained from the wheel encoders using the aforementioned kinematic models [6].
One must be aware of the system's inherent errors, i.e. of systematic (e.g. different wheel diameters, uncertainty in exact position), as well as non-systematic nature (e.g. slipping wheels) [7]. This project implements path integration for the Omnirobot by processing regularly queried data from its wheel encoders. This includes parsing the received events, computing the direction of movement and the covered distance, as well ass visualizing the outcome in a dedicated application.

Learning on ROLLS neuromorphic processor As the final part, this project aimed at implementing, in a simple form, mapping and learning on the reconfigurable on-line learning spiking (ROLLS) neuromorphic processor [12].
Neuronal learning can be implemented on the ROLLS processor using several neuron populations. These desired neuronal groups can be connected with plastic synapses and learning can happen. The learning will be based on spike-timing dependent plasticity (STDP) [8] meaning that the updating of the synapses' weight depends on the time between the pre- and post-synaptic activity. This leads to long-term potentiation (LTP, increase) or long-term depression (LTD, decrease) of the weight [9]. It has been shown that this strengthening of the synapses is sensitive to the correlation of the pre-synaptic activity, which is why this type of learning is called correlation (also Hebbian) learning [10].
In this project, neuronal learning between the robot's location in space and the collision with its surroundings - represented by a neuron population of its own each - was to be shown. Due to the breaking down of the robot, learning between simulated location data and collisions was shown.

2 Methods

2.1 Hardware

Robot The Omnirobot (or Omnibot) is an omni-directional vehicle with three Swedish wheels located at an angle of 120° relative to each other. The device including firmware was provided by NST from TU Munich. It uses an LPCexpresso board with a ARM Cortex-M3 microcontroller from NXP. It is equipped with three Dynamixel MX-28 servomotors , one for each wheel. The servos have a sensor - so-called wheel encoders - that can provide feedback on their current angular position and with this the motors' rotational speed. The device is further equipped with a nine-axis inertial measurement unit (IMU) MPU-9250 from InvenSense and a ring bumper with six sensors around its outer diameter (see figure 1).
The device uses three 11.1 V lithium-ion polymer batteries to power the servomotors. They batteries are further connected to a board converting to 3.3 and 5 V supply voltage for the electronics .
The communication, occurring via a serial interface, is done by streaming strings to the robot's microprocessor upon which the robot could also send back strings (e.g. containing its sensors' current read-out). For this purpose, the microprocessor's embedded UARTs are used. Both USB and Wi-Fi transmission are possible. In this project, only USB was used using a UART-to-USB converter. The current list of commands can be found in the appendix.

Parallella The Embedded Parallella board P1602 (see figure 2) is a high performance computing platform figuring a Xilinx dual-core main processor (Zynq ARM A9) and a 16-core co-processor from Epiphany (E16G301).
The device is running Linux (Ubuntu ARMv7). Connections include 5V power, micro-USB, Ethernet, HDMI and micro-SD. Additionally, there is a direct interface to the ROLLS neuromorphic processor (described below) - i.e. a Paracard board

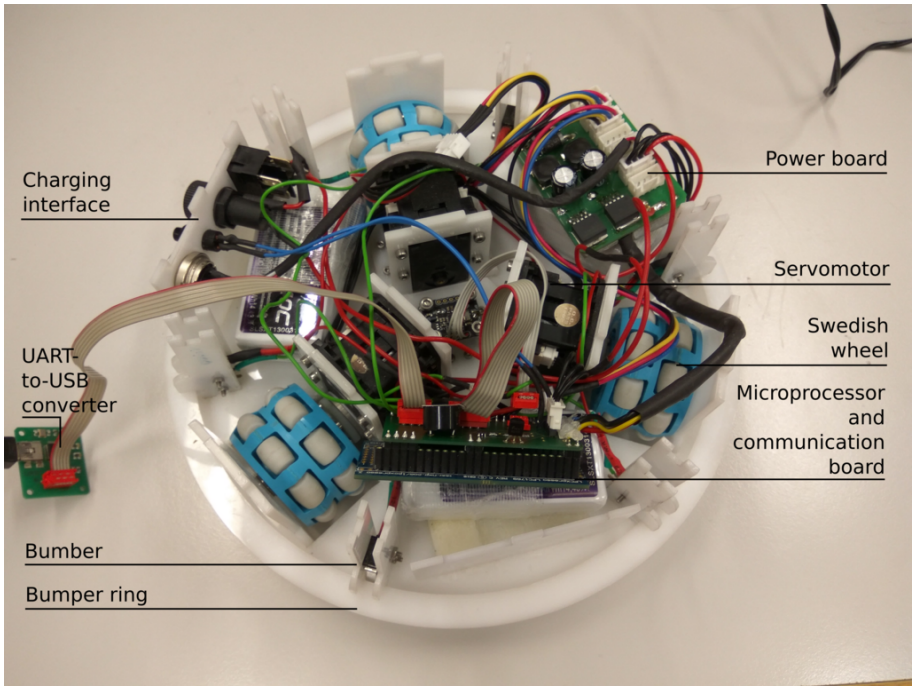

Figure 1: Image of the Omnirobot without top cover

with the ROLLS chip on top is mounted back-to-back via expansion connectors. The Ethernet port allows the user to connect the board to the network and access it remotely via ssh.
During operation, the board gets hot. The temperature in the main processor can reach 95C. This is the reason why it has a large heat sink mounted on top and should be operated with a fan or at least be placed on its side allowing for faster heat dissipation.

ROLLS The reconfigurable on-line learning spiking (ROLLS, see figure 2) neuromorphic processor [8] figures 256 neuron like analog integrated circuits and 131072 synapses (256x256x2).
The neurons are modeled as adaptive exponential integrate-and-fire neurons [11].

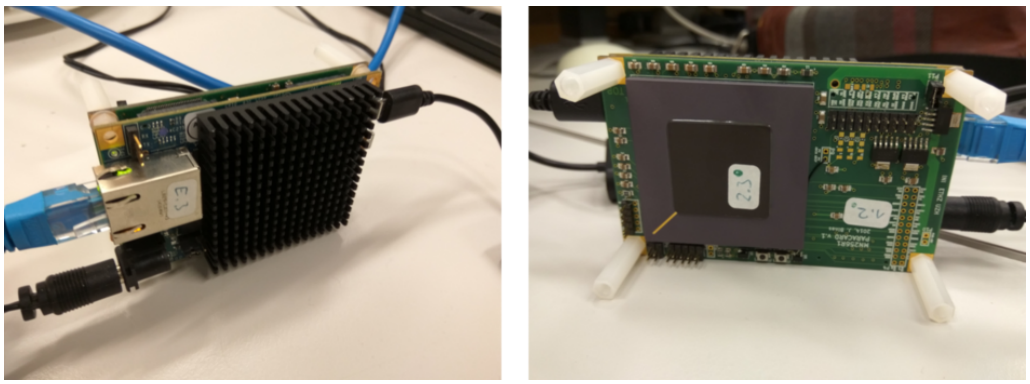

Figure 2: Image of the parallella board during operation (left) and the ROLLS neuromorphic processor mounted on its backside (right)

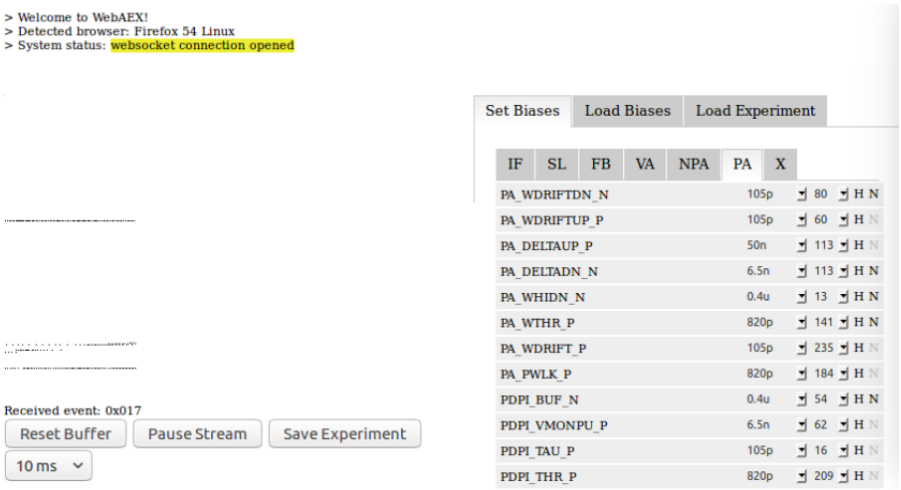

Figure 3: Screenshot of the JavaGui visualizing the ROLLS events during operation

The synapses represent a connection between a neuron-pair. There exist non-plastic (programmable) and plastic synapses, allowing the modeling of both short-term and long-term plasticity mechanisms imitating the actual biological dynamics and enabling on-line learning capabilities. The ability of the neuromorphic chip to emulate learning with biologically accurate dynamics stems from the use of a differential pair integrator (DPI) with its dynamic behavior showing time constants in the range of hundreds of milliseconds.
An additional set of "virtual" synapses allows the application of stimuli from the controlling parallella board. Communication with the chip is based on an address-event representation (AER) protocol, which means events are transferred asynchronously with an address as identifier and a timestamp.
The configuration of the biases for the chip's circuit elements can be done using a Java Web application GUI (see figure3). It is possible to define time-constants, thresholds and a range of other parameters by setting bias currents. On the hardware side these settings are translated from the digital configuration logic via a bias current reference by effectively applying a gate voltage resulting in the desired current. [12].

2.2 Software

NCSRobotLib The NCSRobotLib is a C++ library developed at INI to interface with various devices. It allows for controlling Omnirobot and ROLLS, as well as others such as eDVS and Pushbot. This software environment enables the user to implement multi-agent system with the mentioned devices where data (events) can be shared among them. To do so, a device can be registered as a listener to another one. Upon receiving a new event, a device will notify all its listeners which will then execute a function corresponding to the type of event. The function to be executed is specified for each agent individually (made possible because it inherits from a "DeviceListener.h" implementing a virtual function).

ROLLS/aerctl In order to be able to communicate with the ROLLS chip a software environment had been developed by researchers at INI. It figures a ROLLSDevice.cpp, included in the NCSRobotLib, enabling the creation of the architecture on the neuromorphic chip including the assignment of neuron groups and plastic and non-plastic synapse connection between them. A set of functions collected in aerctl.cpp then handles the communication via the AER protocol.

3 Implementation

3.1 Part 1 - Omnirobot Controller

3.1.1 Kinematics and Wrapper to Microprocessor

In order to be able to properly navigate the Omnirobot and obtain its sensors' values a wrapper to the microprocessor's command list is needed, i.e. several functions based on combinations of microprocessor commands. The most important among them was being able to set its speed in a specified x,y-direction and controlling its orientation. A valuable contribution had already been made by Michel Frising in his semester project "An embedded neuromorphic computing platform for cognitive agents" [14]. In this framework the velocity can be set via an angle Φ_{set}, indicating the direction from the robot's point of view, and a speed v in that direction, superposed with a rotational component $\dot{\theta}_{set}$ (the yaw) resulting in the following representation of the movement with respect to the robot's local frame of reference (see figure 4):

$$\begin{pmatrix} \dot{x}_{local} \\ \dot{y}_{local} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} v * cos(\Phi_{set}) \\ v * sin(\Phi_{set}) \\ \dot{\theta}_{set} \end{pmatrix} \tag{1}$$

However, improvement and updating was needed, because the firmware did not support as many microprocessor commands as were available at that moment. This especially concerned the implementation of the drive command, as setting the servo speeds differed.
The underlying kinematic models [4, 15], presented here, were adapted to the Omnirobot's geometric layout (see figure 4).
The robot wheels' rotational speeds can be expressed with the device's speed in a global two-dimensional system $(\dot{x}, \dot{y})$ and its yaw $\dot{\theta}$:

$$\begin{pmatrix} \dot{\varphi}_0 \\ \dot{\varphi}_1 \\ \dot{\varphi}_2 \end{pmatrix} = 1/r \begin{pmatrix} -sin(\Phi + \alpha_0) & cos(\Phi + \alpha_0) & L \\ -sin(\Phi + \alpha_1) & cos(\Phi + \alpha_1) & L \\ -sin(\Phi + \alpha_2) & cos(\Phi + \alpha_2) & L \end{pmatrix} \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} \tag{2}$$

with r being the wheels' radius (2.54cm), L their distance from the center (8cm), α_i their respective angular displacement (see figure 4) and Φ the robot's heading direction.
The equation for the representation in the robot's local frame of reference reduces

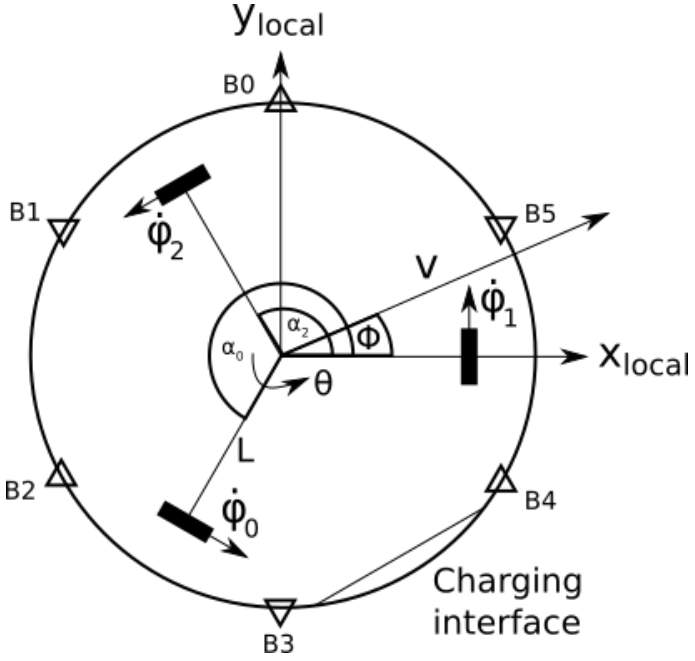

Figure 4: Schematic of the Omnirobot with its wheels, $\dot{\varphi}_i$ and bumpers Bi

to the following when when replacing Φ with 0:

$$\begin{pmatrix}\dot{\varphi_0}\\ \dot{\varphi_1}\\ \dot{\varphi_2}\end{pmatrix} = 1/r \begin{pmatrix}-sin(\alpha_0) & cos(\alpha_0) & L\\ -sin(\alpha_1) & cos(\alpha_1) & L\\ -sin(\alpha_2) & cos(\alpha_2) & L\end{pmatrix} \begin{pmatrix}\dot{x}_{local}\\ \dot{y}_{local}\\ \dot{\theta}\end{pmatrix} \tag{3}$$

In the Omnirobot's case one angle α_0 is subject to definition and the other two will have a value of α_0+120° and α_0+240°, respectively (see schematic in figure 4). Assuming $\alpha_0 = 240°$ (in order to conform with the built-in x and y directions obtained when sending command "!Dx,y,a") this further simplifies into:

$$\begin{pmatrix}\dot{\varphi_0}\\ \dot{\varphi_1}\\ \dot{\varphi_2}\end{pmatrix} = 1/r \begin{pmatrix}\sqrt{3}/2 & -1/2 & L\\ -1 & 0 & L\\ -1/2 & -\sqrt{3}/2 & L\end{pmatrix} \begin{pmatrix}\dot{x}_{local}\\ \dot{y}_{local}\\ \dot{\theta}\end{pmatrix} \tag{4}$$

thus yielding the following equations when inserting equation (1):

$$\dot{\varphi_0} = v/r * sin(\Phi - 240°) + L/r * \dot{\theta} \tag{5}$$

$$\dot{\varphi_1} = v/r * sin(\Phi) + L/r * \dot{\theta} \tag{6}$$

$$\dot{\varphi_2} = v/r * sin(\Phi - 120°) + L/r * \dot{\theta} \tag{7}$$

The required wheel speeds could therefore be calculated. Then, the robot's firmware allowed controlling its servomotors individually by sending the corresponding strings to the microprocessor.
When wanting to express the current speed as a function of the wheels' speed, the following relation can be used:

$$\begin{pmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{pmatrix} = r \begin{pmatrix}cos(30°) & 0 & -cos(30°)\\ -cos(60°) & 1 & -cos(60°)\\ 1/3L & 1/3L & 1/3L\end{pmatrix} \begin{pmatrix}\dot{\varphi_0}\\ \dot{\varphi_1}\\ \dot{\varphi_2}\end{pmatrix} \tag{8}$$

The corresponding global orientation simply results by a coordinate system transformation by the heading angle Φ:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = r \begin{pmatrix} cos(\Phi) & -sin(\Phi) & 0 \\ sin(\Phi) & cos(\Phi) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} cos(30°) & 0 & -cos(30°) \\ -cos(60°) & 1 & -cos(60°) \\ 1/3L & 1/3L & 1/3L \end{pmatrix} \begin{pmatrix} \dot{\varphi_0} \\ \dot{\varphi_1} \\ \dot{\varphi_2} \end{pmatrix} \quad (9)$$

3.1.2 Reacting to Environment

Bumper Ring The robot's bumpers could be used to detect objects and walls in the robot's environment. It consisted of six switches which change their binary state when the surrounding bumper ring pushes towards it and upon release. The bumpers were arranged around the circumference of the robot (see figure 4). Each of the six switches was encoded with one bit putting the bumper value in a range of 0 (no bumper actuated) to 63 (all bumpers activated (theoretical since in practice there is no case in which all switches get turned on at the same time because the ring is stiff)).
The index i of the bumper Bi (see figure 4) corresponds to the number of the bit, starting with bumper B0 being the LSB and B5 the MSB.

Processing The bumper values could be queried from the robot by sending "?Ib" (see appendix). This was done once in every sampling period in a query thread (see figure 7). Then they were read and parsed and stored in a struct with the corresponding timestamp and its time difference to the last bumper event - just like the encoder values which will be explained later in more detail.
In the case of a detected bumper signal other than 0, the robot was given a new direction within a window of 75° opposite to the bumper such that it moves away from the object it bumped into.

3.1.3 Path Integration and Visualization

Path Integration The data required for path integration is the robot's speeds and its heading direction. This can be express with the vector $(\dot{x}, \dot{y}, \dot{\theta})$. Thus, by knowing the robot's wheel speeds φ_i the kinematic relations (8) and (9) may be used for the calculations.
In the ideal case of continuous data no further calculation would be needed. However, with discrete data as is the case here, the covered distance needs to be computed. This works as follows: The path integration function received the current encoder value deltas, as well as their respective timestamp and time delta. With these values, path integration was done by applying the kinematics matrix calculation (see equations 8 and 9) onto the encoder values to find the respective delta in x and y direction and the change in the robot's heading. The result was then added to a globally stored struct containing the previous location of the robot. This whole process was done for local (within the robot's frame of reference) and global coordinates. For the global representation it was assumed that the robot had moved into the same direction during the whole last sampling period. Path integration thus occurred in accordance with the coordinate transform described in equation (9).

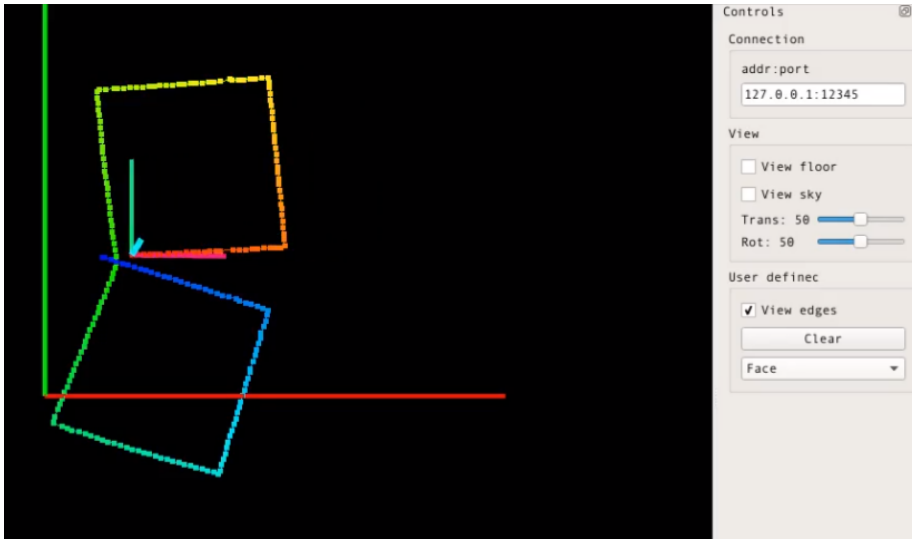

Figure 5: An example of the visualization of the robot's movement in a two-dimensional system (x-axis: red, y-axis:green)

Visualization In order to be able to track and record the robot's movement some sort of visualizer was desired. For this project, an already existing visualizer application based on Qt widgets was used. It was provided by Julien Martel and consisted of a receiver and a transmitter. To have the desired data transferred, a few alteration to the protocol for the communication of those two had to be made (e.g. defining what type of data was to be sent). Furthermore, since the Omnirobot is moving in a two-dimensional system only, the viewing directions of the visualizer were adjusted (see figure 5 for an example).
The data which the transmitter received was the result of the path integration. For longer intervals without new data, the previous direction and speed was assumed.

3.1.4 Velocity Controller

This part of the project aimed at implementing a standard PID controller [2,3] for the Omnirobot's velocity. This was done by using the wheel encoder values as a

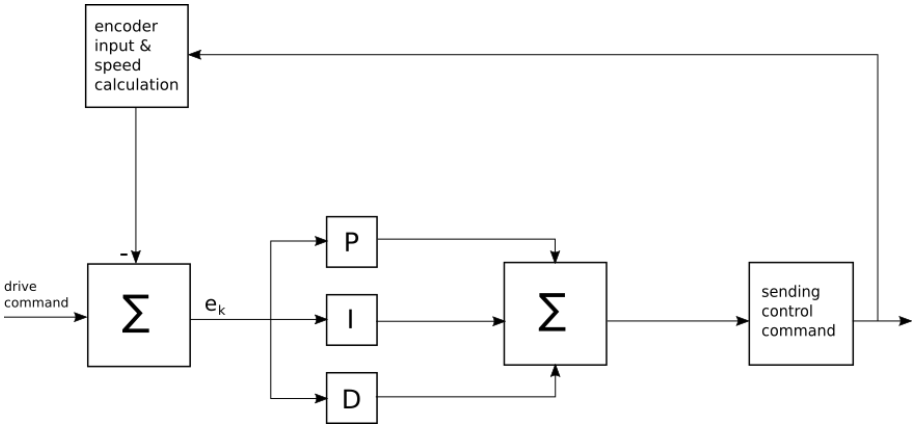

Figure 6: Schematic of the PID controller implemented in this project

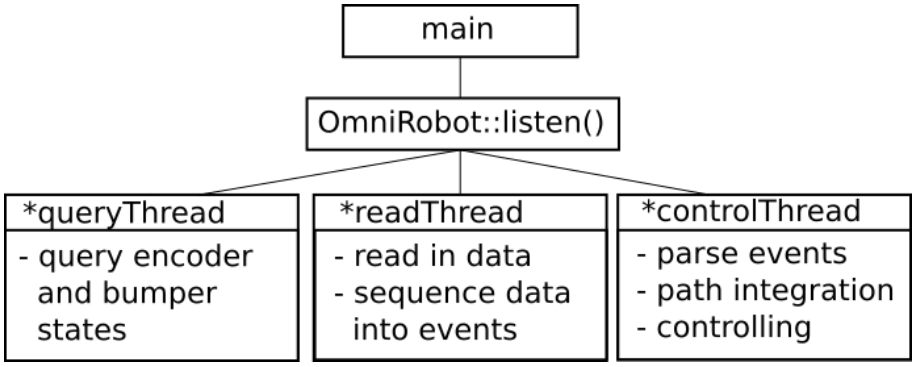

Figure 7: Schematic of the thread setup used for data acquisition and controlling

feedback which allowed the computation of the current speed and its difference to the set value which was stored and updated each time new drive commands were sent. A block diagram of the controller architecture is shown in figure 6.
To do so, a series of threads was used (see figure 7). One thread (denoted queryThread) would simply loop and query the robot with the "Get encoder values" command ("?PA") on a regular basis. This sampling time was defined to be 50 milliseconds. The effective implementation was a call of a write function to the file identified by the filedescriptor corresponding to the Omnirobot's USB port. This function transferred the corresponding strings to the microprocessor.
A second thread was run to perform the reading of the serial bus when available. For this, a read function was called which would read out characters from the serial bus and store them in a character array.
In the controlling thread, the sequenced events were processed. This contains parsing according to their respective type (buffer and encoder events were treated). The events would get stored in a private struct. Upon finished parsing of encoder events, they were further processed. In a first step, the respective difference to the previous encoder values and timestamp was calculated. Then, the corresponding wheel velocity was computed from these two values. This velocity was compared to the most recent set value to compute the error. This error was then multiplied and integrated and used to calculate the control command to be sent to the robot.

3.2 Results Part 1

In order to be able to estimate the error in the robot's movement with and without the controller, a measurement series was executed. For this purpose, the robot was moved in a certain pattern in an "arena" with distances marked on the ground. The driven shapes were a square, a triangle and a circle. For the square and the triangle the shape was repeated three times per measurement sequence. The sequences were recorded with a handheld Sony digital camera mounted over the arena. Plus, the resulting path integration data was logged (see figure 8). Then, the average speed, as well as the offset at the end of each sequence, was measured by eye using a media player and the written scales.
This was done for the robot without the controller, as well as with a controller and different parameters. For the circle only the resulting offset upon completion was

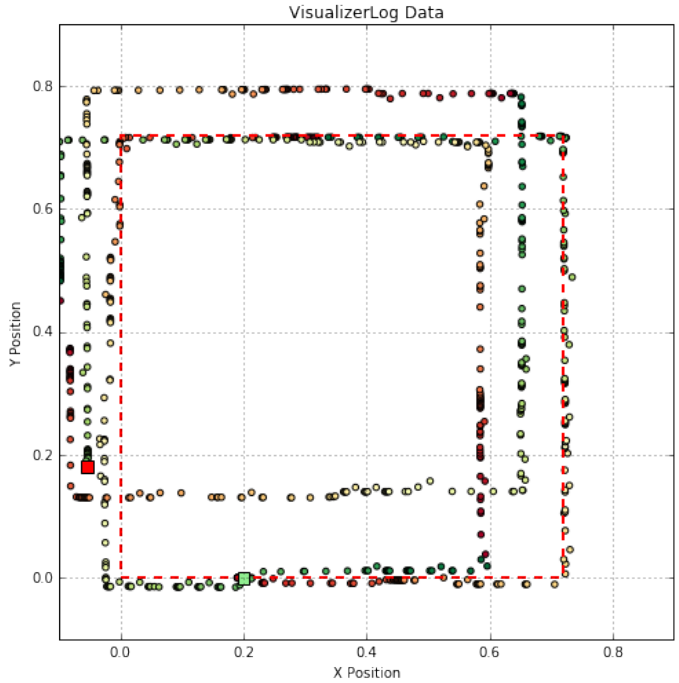

Figure 8: Recorded pattern logged by the visualizer based on the path integration data it had received. The driven sequence was three squares. The expected pattern is shown as a red dashed box. Logging started at the light green marking and ended at the red one.

measured as it was not feasible to determine the covered distance accurately enough due to the non-linear movement. The results are presented in tables 1 and 2.
At that point, the controller still took into account values for all timestamps which was one reason why it performed very bad. Unfortunately, due to the robot's microprocessor becoming non-functional no further tests could be conducted. However, the robot had a sufficiently low error in movement even without the controller.

3.3 Discussion and Outlook Part 1

A major issue faced was that data available to the controller were of the form seen in figure 9. During most of the time, for durations of up to over one second no new

Table 1: Offset analysis

Speed	Sequence	Δx [cm]	Δy [cm]	$\Delta\theta$	Distance covered [cm]
25%	3x square	0	-1	<5°	213
25%	3x square (with controller)	5	12	80°	213
50%	3x square	0	-1	<5°	277.5
50%	3x triangle	3	0	<5°	205.5
50%	1x circle	1	-1	<5°	144
50%	1x circle	1	-1	<5°	144

Table 2: Average speed analysis

Speed	Sequence	Avg. speed [cm/s]	Ref speed [cm/s]	Error in %
25%	3x square	5.92	5.90	0.3
50%	3x square	11.55	11.81	-4.8
50%	3x triangle	11.24	11.81	-2.2

values (i.e. no character string at all) were present.
There are several possibilities that could lead to such a lack of data: The write or read operation are not executed as desired, data is not sent or transmitted within that time or data acquisition is not done properly.
It can be noted that when data points were received they seemed to be of the correct current encoder state - except for a few exceptions. This assumption is based on the fact that for a continuous turning of the wheel with no acceleration the data points follow the expected linear curve in the range 0 to 4096 (see figure 9). It is therefore believed that some kind of delay must be present somewhere in the system.
When taking a closer look at the data distribution it can be seen that the density of data points is higher after the periods without data than right before. This can also be confirmed when looking at the computed time difference between those data points which was in the order of a few milliseconds instead of the expected 50 ms. In order to further investigate this behavior, it was checked and affirmed that the write commands are sent in a regular fashion - i.e. without any similarly large delays. A test with the robot's stream mode - within which the microprocessor sends the desired data in a regular fashion without the need of query commands - was also conducted but showed the exact same issues. Furthermore, even when using a very simple program using only a read and write function each in one single thread did not yield different results either.
Then, by evaluating the elapsed run time of the read function it could be seen that it does not return for comparable time durations. When no data is present at the port, read() would return immediately. It is thus more likely that the read function has received some (possibly invalid) data but no endline character which leaves it "hanging", i.e. waiting for more data to arrive. This would mean that the data acquisition or transmission from the Omnirobot is showing a delay or malfunctioning. When looking at the robot's side, various sources could be the reason for this, most

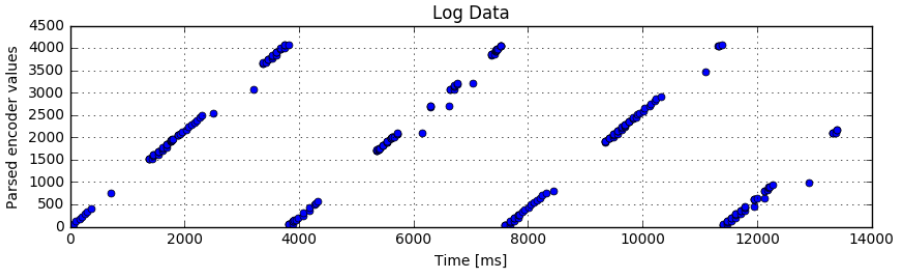

Figure 9: Visualization of parsed encoder values and their corresponding time of arrival. The maximum value is 4096. Conducted at constant speed for wheel 0.

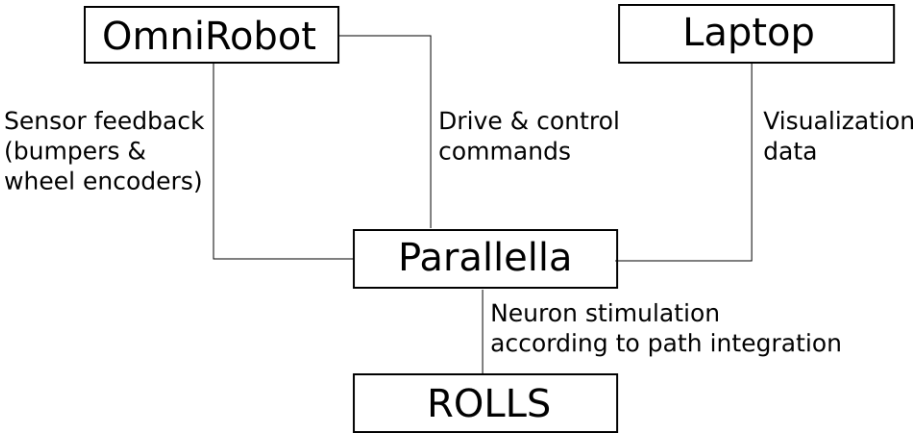

Figure 10: Schematic of the interfaces intended to be used in this project

of which cannot be determined because of the lack of knowledge of both its firmware and hardware. However, there is evidently a timeout for reading the wheel encoder's values. This was discovered when investigating faulty encoder values (-128) which always lead to an increased response time in the order of 30 ms. Consequently, the delay does not come from the encoder readout and is likely to stem from data processing or communication channels.
This obviously posed a serious and unsolvable problem for the controller, as during this time, no feedback can be obtained and thus no controlling was possible. Furthermore, at the beginning, the very small time differences between events arriving after a period with no data lead to the calculation of false speeds. This was due to the computation time having a too high influence on the result. Because with such small time-deltas even small deviations from the correct value resulted in a large error when executing the division for the speed calculation.
After the discovery of this issue and in order to still get the robot running with the controller, events that arrived with timestamps below and above a certain threshold (as a percentage of the sampling time) were neglected (i.e. the speed set by the last drive command was assumed so it would not lead to an error that gets processed by the controller).
What was even more unfortunate is the fact that the robot broke a couple of weeks before the end of the project. From the investigations performed it can be assumed that the microprocessor board is broken (the power board still produces the desired voltages and the servomotors are able to set the torque).
This lead to unfinished testing of the controller (e.g. the controller's performance when only using potentially valid data) and no tuning of its parameters could be conducted.
The lack of continuous data obviously also affected the performance of the visualizer and the interaction of the robot with its environment. On the one hand, the visualizer application would experience jumps from one data point to the next with a great distance between them and can therefore not indicate the robot's position at some points in time. To work around this issue, it was decided to generate location data points when no new event arrived for a certain period of time (defined as twice the sampling period). This was done by assuming the same movement as calculated

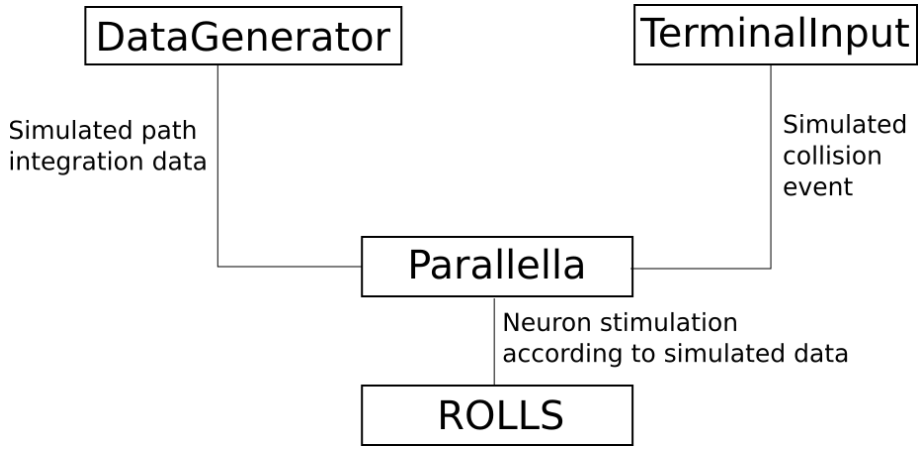

Figure 11: Interfaces as used for the simulation

for the last data point.
On the other hand, the time for recognizing a bumper event had occurred, could be delayed by up to one second owing to the delay. This makes the system rather slow in reacting to its environment. It also means that the stimulation of the bumpers below one second could go unnoticed by the system, because no data acquisition took place in that time.
Furthermore, apparently bumper 4 was broken and could therefore not serve for providing feedback. The reason remains unknown. The electrical connection to the switch seem to be intact.

3.4 Part 2 - Simple Map Building on the ROLLS neuromorphic processor

3.4.1 Localization and Mapping using the Omnirobot

Combining path integration performed on the parallella and the feedback from the bumpers it is possible to implement a simple SLAM algorithm taking advantage of the properties of the ROLLS neuromorphic chip. For this, an architecture with three populations of neurons was defined on the ROLLS processor: Two with 64 neurons each to represent the robot's position in space and a third figuring 16 neurons standing for collisions with the surrounding. Upon a change in the robot's location or the state of its bumpers - i.e. when a new event arrived - the ROLLS controller running on the parallella and registered as a listener to the Omnirobot would stimulate the corresponding neurons. By defining plastic synapses between the location and the collision neurons, the coordinates of such a collision could be learned. The proposed architecture using also the controller and the visualization is shown in figure 10.

3.4.2 Learning with simulated data

Due to the robot becoming non-functional, alternatives to the described task had to be sought. In order to still use the existing framework and the ROLLS processor

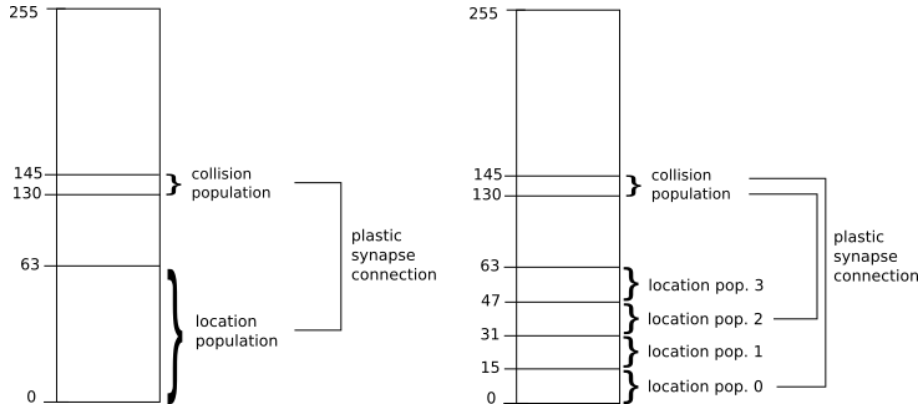

Figure 12: Illustration of the architectures used for the ROLLS experiments. During the first experiment, one location population was used (left). It was later increased to four (right) in a second experiment.

to show learning, it was decided to use simulated location and collision data. This resulted in the need for slightly changed interfaces in accordance with figure 11.
For a first experiment, the neuron architecture shown in figure 12 on the left was applied, meaning 64 neurons (No. 0 to 63) were used to represent the simulated location. Another group of 16 neurons was stimulated when a simulated collision took place (explained later). A connection with plastic synapses was established from the location neuron group to the collision one, thus enabling learning between the two.
A 'data generator' was used to simulate a simple one-dimensional environment. It counted up and down within a finite interval (0,63) with a fixed increment and at a defined frequency (50 ms). This effectively generated a simulated location point with each iteration, mimicking the results of path integration. After each counting step, a warn event was issued to the DataGenerator's listeners (in this case the ROLLS controller). There, this data was used to stimulate the corresponding neuron on the neuromorphic processor.
For the simulation of the collision event the existing TerminalInputDevice.cpp was adapted to react to a specific key input (chosen to be "w"). This key could be actuated manually when in the active terminal, and served as the initiation of the simulated collision. Upon that, the entire collision population of neurons on ROLLS was stimulated. During this entire process, all events could be logged and saved with a timestamp.
In order to obtain good and reproducible learning results, adequate biases for the ROLLS processors needed to be selected. It was decided to start with a modified version of the 'plastic' biases provided by Raphaela Kreiser (see appendix).
A total of three different experiment series were conducted. For the second one, four instead of one location population were used (see figure 12, left). Only two of them were connected to the collision population, meaning only they were able to learn a connection. This was employed to achieve spatial resolution of the learned connection. Reasons for this measure will be explained below.

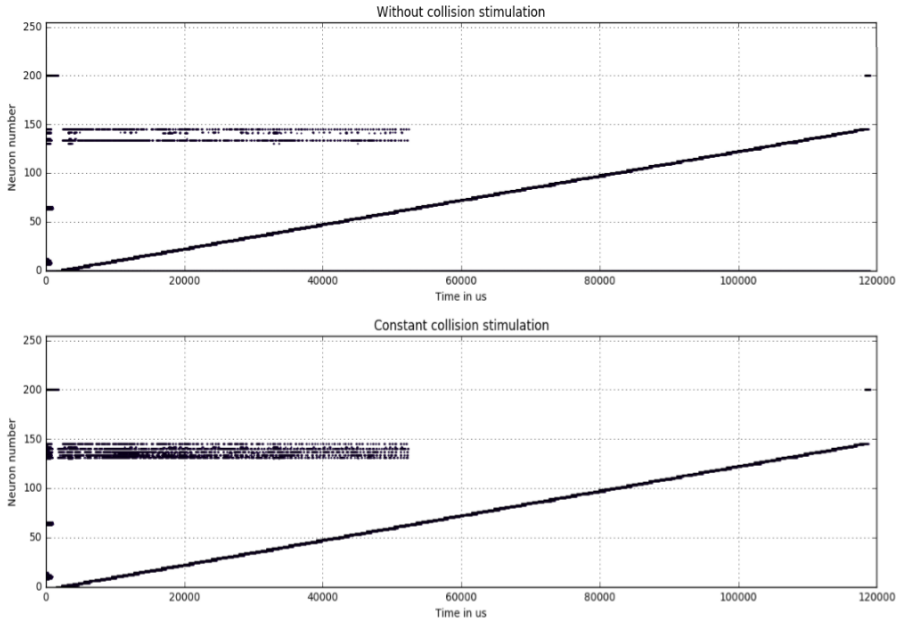

Figure 13: Visualization of active plastic synapses with stimulation (bottom) and without stimulation (top) between neurons

3.4.3 Results Part 2

For the evaluation of the first experiment, a program provided by Raphaela Kreiser was used. It enabled reading out the state of the plastic synapses after an experiment run had been completed. For this, the ROLLS controller had to be switched off and the plastic synapse connections between the neurons were disabled. Then, each neuron and its plastic synapses were stimulated iteratively while logging the received events with their timestamps. Each iteration takes place within 700 milliseconds. The resulting plot thus represents the sequentially stimulated neurons as a continuously increasing line. Within the same time window, the (pre-synaptic) neuron's plastic synapses are stimulated, thus enabling invoking a response of the post-synaptic in the case of an active plastic synapse. This directly translates into a learned connection. Hence, each neuron active during the same time window as a neuron on the continuous line can be interpreted as an active plastic synapse.
The results of the first experiment are shown in figure 13. Due to non-ideal biases at the time, even without any simulated collision some collision neurons are active (figure 13, top). This can be interpreted as a noise floor concerning the active synapse connections. Learning can thus only be shown in comparison with this noise floor. The bottom plot shows the same populations after a constant stimulation was executed by continuously pressing the correct key in the terminal. Clearly, a greater number of neurons fire, therefore proving that a connection between the two population had been learned.
The issue that presented itself during this experiment was that the same effect could be observed upon simulating a collision only for a short time. In that case, only the synapses between the concurrently active location neurons should have learned and not all of them.

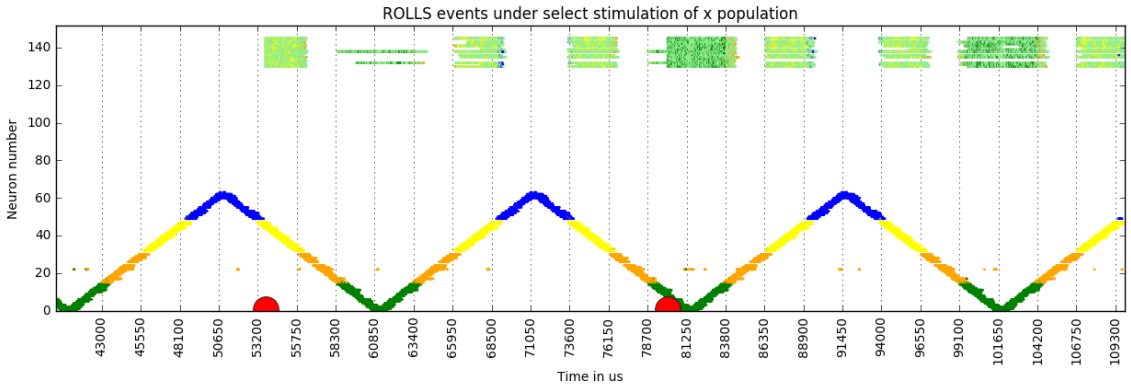

Figure 14: Plot of the events sent by the ROLLS during experiment 2. Proof of learning occurring on the ROLLS chip. After stimulation (start indicated by red dot) of the collision population (light green, neurons 130 to 145) it gets reactivated every time the corresponding position (triangles between 0 and 63) gets stimulated again. The four different location populations are colorized for better understanding).

It is argued here that an endless learning loop is formed: Once the collision population is stimulated externally by pressing a key, the connection between the currently active location neuron and the collision population is learned. These neurons keep firing as long as the location neuron is active. It was in the nature of the data generator's design that the next location neuron will get activated while the previous neuron is still active. Due to the collision population still being potentiated - because of the learned connection with the previous neuron - this plastic synapse will also learn, even though no collision is actually occurring anymore. This effect cannot stop and consequently all plastic synapses will be active after one iteration over the whole space.
This is why in a second experiment, conducted on Raphaela Kreiser's parallella, it was decided to split the one location population into four (see figure 12).
With only two non-neighboring groups being connected to the collision population, it was possible to stop the above described effect by giving the collision neurons enough time to stop firing before learning with the next location neuron would (falsely) happen. Figure 13 proves the forming of a learned connection between two populations upon a simulated collision. This can be interpreted as a very simple map with a simulated wall or object at locations represented by the two groups 0 and 2.
It is therefore successfully shown, that with the proposed setup learning between individual neuron groups can be achieved on the ROLLS neuromorphic processor.
Note that the collision population remains potentiated for a short period of time even after the corresponding pre-synaptic neurons in the location population have stopped firing. This speaks in favor of the before proposed explanation of an endless learning phenomenon. By using a different neuronal architecture in experiment two, this effect could be interrupted and a very basic spatial resolution of the learned collision could be achieved.
In order to further improve this resolution, the simulation was slightly adapted for a third experiment, also conducted on Raphaela Kreiser's parallella. Upon the arrival

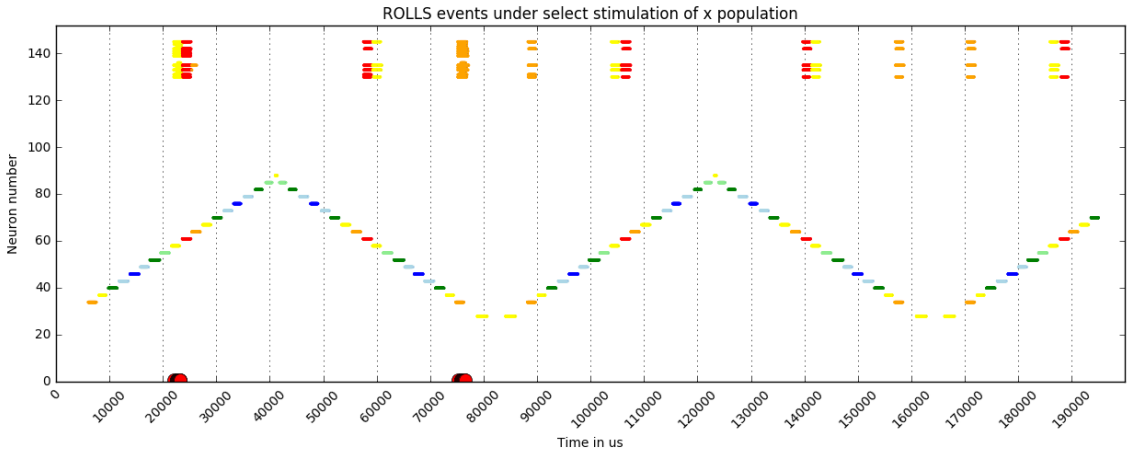

Figure 15: Plot of the neuron events sent by ROLLS during experiment 3. For better understanding, each neuron was assigned a different color and any collision event received within the same time frame as a location neuron got the same one. Simulated collision events are marked at the bottom of the plot with a red dot.

of a new event, a neuron would get stimulated many times with the total stimulation duration averaging between 1 and 2 seconds. The main difference was that the stimulation of a new location neuron would be delayed by a certain time. This was defined to be 750 milliseconds. The goal of it being to give the collision neurons time to cease firing and hence stop the aforementioned endless learning loop. Additionally, only every third neuron was stimulated. Furthermore, for this experiment the biases "sequence2_3" were used, because for this particular case the ones used before resulted in a permanent potentiation of the collision neurons. This might be due to a difference in temperature resulting in altered behavior. It was later repeated and similar results could be obtained. The results are shown in figure 15. Note that for this experiment, the location population was shifted up by 25 neurons to avoid any interference by neuron 22 which was constantly firing, even though it was stimulated at all. For the plot it was neglected. Additionally, neurons 137 and 138, in the collision population, falsely learned connections and kept firing, which is why they are also ignored in the final plot. The same effects could already be seen in experiment two (see figure 14).
The results prove the successful learning of two positions of collision events, now using only a single neuron group for the location. The first collision occurs while neurons 61 and 64 are active (red and yellow). The renewed activation of these two neurons invokes a short potentiation of the collision neurons, thus successfully showing the plastic synapses being active. The same could also be shown a second time with even just a single neuron (34) having formed the connection.

3.4.4 Discussion and Outlook Part 2

The successful learning of a simulated location to a collision population is a proof of concept for the mapping using the ROLLS neuromorphic processor and enables a series of application to be performed (discussed later).
The results of the experiment may be improved by further tuning the ROLLS biases

and the delay and frequency of stimulation upon the arrival of a new location event. Furthermore, changing the frequency and number of stimulation of the collision population might yield more plastic synapse connections being active, hence providing a clearer signal. It should also be attempted to get the plastic synapses connected to neurons 137 and 138 to only learn when they have to. This is most likely to be achieved by changing the ROLLS biases, but shifting the group to a different set of neurons might also work, as different neurons show slightly different behavior.
In a next step, one could rather easily extend the simulation to two dimensions using a third population representing y-locations.
It should further be aimed at getting rid of the falsely firing neuron 22 as this could interfere with the learning in case it is assigned a plastic synapse connection.
There currently exist two parallella boards with one ROLLS processor each. The best and last experiment was conducted on Raphaela's parallella after it failed to work on the other one. It is likely that the exact same biases will lead to a different or no result at all on the other ROLLS. This statement remains to be verified in the future.

4 Final Discussion and Outlook

The work done in this project laid the foundation for controlling a omni-directional robotic vehicle. Given a functional Omnirobot device is available, a characterization of the controller could be pursued - i.e. for periods where data with valid timestamps is available. There is even a chance that with the new microprocessor data streams will be continuous at all times. In that case, the controller's parameters can be tuned in order to find a good performance. It can also be optimized by increasing the sampling rate. So far, measurements were conducted at 50 milliseconds sampling time only. However, 10 milliseconds are realistic. Further decrease would likely require parallelization of the parsing and path integration computations.
Given the low error in movement of the robot, controlling of the robot might not be a top priority - at least at low to moderate speeds.
In a second step the learning described in this report could be applied using the events provided by the parsing and path integration functions from the Omnirobot class as described in part 2 of this project. This should also be easily extensible to two dimensions, enabling map building of the robot's actual surrounding. Given learning is successfully and reliably working one could then use the active plastic synapse connections to predict imminent collisions and act accordingly. Possibilities therefore include obstacle avoidance and a bit more advanced (though still basic) SLAM.
The software environment in place also allows for the extension of the system with devices such as the eDVS enabling the use of vision to improve the performance of the setup.

5 Final Conclusion

In this project, a PID controller for an omni-directional robotic vehicle was attempted to build. However, due to delayed data responses from the robot, it could not have been considered functional. Proper characterization of the controller was made impossible due to the device's microcontroller break down.
For the second part, learning on a reconfigurable on-line learning spiking (ROLLS) neuromorphic processor between two populations of neurons could successfully be shown. It is therefore possible to learn the location of a collision on hardware. This serves as a proof of concept showing that simultaneous localization and mapping of a robotic vehicle is possible using sensory data (if available) and the ROLLS neuromorphic processor enabling learning.

Acknowledgments

Many thanks go to Yulia Sandamirskaya for helping me with this project and providing valuable feedback and ideas. I also thank Tobi Delbrück for making this project possible.
Special thanks go to Raphaela Kreiser for her great and patient help with the ROLLS biases.
I further thank Julien Martel for providing the visualizer application and anyone who helped develop the NCSRobotLib.

References

[1] D. J. Balkcom *et al.*, "The time-optimal trajectories for an omni-directional vehicle," *The International Journal of Robotics Research*, vol. 25, no. 10, pp. 985–999, 2006.

[2] F. Ribeiro *et al.*, "Three Omni-Directional Wheels Control on a Mobile Robot," *Control 2004*, 2004.

[3] X. Li and A. Zell, "Motion control of an omnidirectional mobile robot," *Informatics in Control, Automation and Robotics*, 2009.

[4] P. F. Muir and C. P. Neuman, "Kinematic modeling of wheeled mobile robots," *Journal of Robotic Systems*, vol. 4, pp. 281–340, April 1987.

[5] D. Lambrinos *et al.*, "Mobile robot employing insect strategies for navigation," *Robotics and Autonomous Systems*, vol. 30, no. 1, pp. 39–64, 2000.

[6] J. Inthiam and C. Deelertpaiboon, "Self-localization and navigation of holonomic mobile robot using omni-directional wheel odometry," October 2015.

[7] J. Borenstein *et al.*, "Mobile robot positioning: Sensors and techniques," *Journal of Robotic Systems*, vol. 14, pp. 231–249, April 1997.

[8] N. Qiao *et al.*, "A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128K synapses," *Frontiers in Neuroscience*, vol. 9, p. 141, April 2015.

[9] L. F. Abbott and S. B. Nelson, "Synaptic plasticity: taming the beast.," *Nature neuroscience*, vol. 3, pp. 1178–1183, November 2000.

[10] S. Song *et al.*, "Competitive Hebbian learning through spike-timing-dependent synaptic plasticity.," *Nature neuroscience*, vol. 3, pp. 919–926, September 2000.

[11] G. Indiveri, E. Chicca, and R. Douglas, "A VLSI array of low-power spiking neurons and bistable synapses with spike-timing dependent plasticity," *IEEE Transactions on Neural Networks*, vol. 17, pp. 211–221, jan 2006.

[12] T. Delbruck, R. Berner, P. Lichtsteiner, and C. Dualibe, "32-bit configurable bias current generator with sub-off-current capability," in *ISCAS 2010 - 2010 IEEE International Symposium on Circuits and Systems: Nano-Bio Circuit Fabrics and Systems*, pp. 1647–1650, IEEE, may 2010.

[13] A. Dietmüller and H. Blum, "Obstacle avoidance and target acquisition with an event-based camera on a neuromorphic chip," 2017.

[14] M. Frising, "An embedded neuromorphic computing platform for cognitive agents," 2016.

[15] M. I. Ribeiro and P. Lima, "Kinematics models of mobile robots," 2002.

A Omnirobot Command List

```
Commands:
 ?I[g/a/c/b/A]   - get (g)yro, (a)ccelero, (c)ompass, (b)ump, or (A)ll
 !I[1/0],f,data  - Set SD Stream: On/Off, frequency, data
                 - where data is a bitmask with the following bit to sensor
correspondence:
                 - bumper 0, wheel speed 1, gyrometer 2, accelerometer 3, euler
angles 4, compass 5
                 - servo voltage 6, servo position 7, servo temperature 8, servo
load 9

 !E[0,1,2]       - 0: All answ. on(default), 1: Cmd Echo off, 2: All answ. off

 !Dx,y,a         - drive towards x(fwd), y(swd); a(rot) [-70...70]
 !DDx,y,a        - same as !Dx,y,a but with decay after 1s
 ?D              - get x,y,rot

 ?B              - get battery Voltage
 ?Ca             - ping servo with id a

 !Sa=b           - set servo id a to id b
 !S=b            - set servo id to b

 ?L[0-16]        - get servo load
 ?LA             - get all servos load

 ?Mid,address,len - read servos memory. Returned data is hexadecimal
 !Wid,address,len,bytes - write servo memory. Bytes have to be specified as
                    decimal integers separated by commas.

 !P[0-16][s]     - set servo speed
 !PA[s]          - set all servos speeds

 ?S[0-16]        - get servo speed
 ?SA             - get all servos speed

 !G[0-16]        - set servo goal position (in joint mode)

 ?P[0-16]        - get servo position
 ?PA             - get all servos positions

 ?T[0-16]        - get servo temperature
 ?TA             - get all servos temperatures

 !T(0|1)[0-16][t] - enable/disable and set servo torque

 ?V[0-16]        - get servo voltage
 ?VA             - get all servos voltages
 R               - reset board
 P               - enter reprogramming mode

 ??              - help
```

B ROLLS biases used for this project (Omnibot-Plastic2)

IF | SL | FB | VA | NPA | PA | X

IF_RST_N	15p	17	H	N
IF_BUF_P	50n	56	H	N
IF_ATHR_N	15p	0	H	N
IF_RFR1_N	105p	240	H	N
IF_RFR2_N	105p	240	H	N
IF_AHW_P	15p	0	H	N
IF_AHTAU_N	15p	9	H	N
IF_DC_P	15p	0	H	N
IF_TAU2_N	105p	192	H	N
IF_TAU1_N	105p	100	H	N
IF_NMDA_N	15p	17	H	N
IF_CASC_N	15p	17	H	N
IF_THR_N	105p	131	H	N

IF | SL | FB | VA | NPA | PA | X

SL_THDN_P	0.4u	52	H	N
SL_MEMTHR_N	6.5n	25	H	N
SL_BUF_N	0.4u	73	H	N
SL_THMIN_N	105p	32	H	N
SL_WTA_P	0.4u	73	H	N
SL_CATHR_P	0.4u	255	H	N
SL_THUP_P	24u	255	H	N
SL_CATAU_P	15p	45	H	N
SL_CAW_N	0.4u	255	H	N

IF | SL | FB | VA | NPA | PA | X

FB_REF_P	15p	0	H	N
FB_WTA_N	15p	0	H	N
FB_BUF_P	15p	0	H	N
FB_CASC_N	15p	0	H	N
FB_INVERSE_TAIL_N	15p	0	H	N
FB_INVERSE_REF_N	15p	0	H	N

IF | SL | FB | VA | NPA | PA | X

VA_EXC_N	24u	199	H	N
VDPIE_TAU_P	105p	10	H	N
VDPIE_THR_P	820p	200	H	N
VA_INH_P	24u	255	H	N
VDPII_TAU_N	105p	10	H	N
VDPII_THR_N	820p	100	H	N

IF | SL | FB | VA | NPA | PA | X

NPA_PWLK_P	820p	228	H	N
NPA_WEIGHT_STD_N	105p	17	H	N
NPA_WEIGHT_EXC_P	105p	168	H	N
NPA_WEIGHT_EXC0_P	0.4u	211	H	N
NPA_WEIGHT_EXC1_P	0.4u	193	H	N
NPDPIE_THR_P	820p	168	H	N
NPDPIE_TAU_P	105p	208	H	N
NPA_WEIGHT_INH_N	820p	172	H	N
NPA_WEIGHT_INH0_N	6.5n	161	H	N
NPA_WEIGHT_INH1_N	50n	52	H	N
NPDPII_TAU_P	15p	43	H	N
NPDPII_THR_P	6.5n	130	H	N

IF | SL | FB | VA | NPA | PA | X

PA_WDRIFTDN_N	105p	80	H	N
PA_WDRIFTUP_P	105p	60	H	N
PA_DELTAUP_P	50n	113	H	N
PA_DELTADN_N	6.5n	113	H	N
PA_WHIDN_N	0.4u	13	H	N
PA_WTHR_P	820p	141	H	N
PA_WDRIFT_P	105p	235	H	N
PA_PWLK_P	820p	184	H	N
PDPI_BUF_N	0.4u	54	H	N
PDPI_VMONPU_P	6.5n	62	H	N
PDPI_TAU_P	105p	16	H	N
PDPI_THR_P	820p	209	H	N