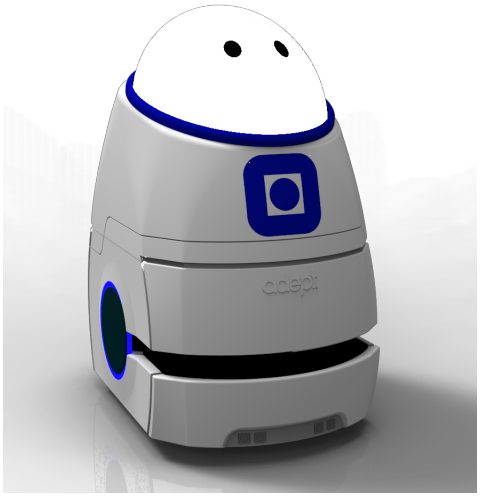
The NTNU Cyborg

Exploring Visualisations and Behaviour

Johanne Døvle Kalland



Supervisor:
Associate Professor Sverre Hendseth
Assisting Supervisor:
PhD Candidate Martinius Knudsen

TTK4550 - Engineering Cybernetics, Specialisation Project
Department of Engineering Cybernetics
Autumn 2019

Task Description

The Cyborg project is currently undergoing work that when done will allow the robot to drive autonomously around in Glassgården at NTNU, while showcasing visualisations of the neural activity of the nerve-cells at St. Olav's Hospital. It may also run several different animations depending on its current mood. The work in this project focuses on improving software and documentation of existing work, as well as examine possible behaviours to be implemented on the Cyborg. The student shall:

- Reevaluate the Cyborg as it was at the start of this work.
- Re-frost the LED-dome so the internal LEDs do not show.
- Improve the documentation of the Cyborg, therein collect all code into one repository on GitHub and start work on updating the Cyborg's wiki.
- Examine possibilities for behaviours to be implemented on the Cyborg, including animations on the LED dome, physical movement and moods.
- Explore strategies on how to work with Legacy code, and the consequences of those.

Preface

This project is written at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology for the NTNU Cyborg project in the course TTK4550 - Engineering Cybernetics, Specialisation Project. The Cyborg project is developing a robot connected to live neural cells grown at St. Olav's Hospital in Trondheim. The purpose of this work has been to continue the development of the cyborg, especially the LED-dome and the behaviour.

I would like to thank Sverre Hendseth for his guidance in writing this report, and Casper Nilsen and Lasse Göncz for being great coworkers on the project, and people to bounce ideas o of.

Abstract

The goal of this report has been to gain knowledge about the cyborg in the NTNU Cyborg project, both for the work done this semester and for a future masters thesis. The NTNU Cyborg project aims to create a robot able to communicate with living nerve tissue, which will then be able to serve as a mascot for the university. A big part of the focus of this work has been prestudies, entertaining possible ideas and reading code, not so much implementing new software. At the beginning of my work, the cyborg was already far along to becoming a functioning mascot, though there is work yet to be done. Some of that work done this semester has been to implement two new animations, as well as making the LED-dome of the cyborg less transparent, in addition to several smaller improvements.

As the project is being worked on by students for their specialisation project and masters theses, proper documentation of the work being done is essential. Work on a wiki collecting all relevant documentation has been started. The reason being not having to read through the abundance of reports written through the roughly five years the project has been active. There already exists one general and more superficial wiki. The new is more directed towards the software. Starting work on a project that has run for many years is not easy. As such, in the interest of making it easier for future me and other students, part of the task was to examine methods of working with legacy software and what legacy software is.

Contents

Task Description i

Preface ii

Abstract iii

1 Introduction 1
- 1.1 The NTNU Cyborg Vision . . . 1
- 1.2 Previous Work with the NTNU Cyborg . . . 2
- 1.3 Other Ongoing Work . . . 3
  - 1.3.1 Replacing the Navigation Stack . . . 3
  - 1.3.2 Creating a GUI website . . . 3
- 1.4 Work not mentioned any further . . . 4

2 Background 5
- 2.1 Hardware . . . 5
  - 2.1.1 Cyborg Base - Pinoeer LX . . . 5
  - 2.1.2 LED-Dome . . . 6
  - 2.1.3 Mode Selector Box . . . 6
  - 2.1.4 MEA2100 - System Micro-electrode Array . . . 7
  - 2.1.5 NodeMCU ESP-32S . . . 7
- 2.2 Software . . . 8
  - 2.2.1 FreeRTOS . . . 8
  - 2.2.2 Arduino IDE . . . 10
  - 2.2.3 Software with the Pioneer LX Base . . . 10
- 2.3 ROS - Robotic Operating System . . . 10
  - 2.3.1 The Concepts of ROS . . . 11
- 2.4 SMACH - State MACHine . . . 13
  - 2.4.1 Creating a SMACH State Machine . . . 14
- 2.5 Cyborg Software Structure . . . 15

3 Legacy Software 19
3.1 The Definition of Legacy Software 19
3.2 Approaching Legacy Code 19
3.2.1 Unfamiliar Code 20
3.3 The Mechanics of Change 21
3.4 Changing Software 22
3.4.1 Changing the Code Quickly 23
3.4.2 How to Add a Feature 23
3.5 Dependency-Breaking Techniques 24

4 Reevaluating the Cyborg 26
4.1 Previously Proposed Tasks 26
4.2 Discussion of Proposed Tasks 28
4.2.1 The Cyborg in General 29
4.2.2 The State Machine 30
4.2.3 Behaviour 30

5 Visualisations 32
5.1 Battery Visualisation 32
5.2 Roadwork Visualisation 35

6 Minor Additions to the Cyborg 37
6.1 Udev Rules for Accessing USB Connections 37
6.2 Creation and Update of Wiki on GitHub 38
6.3 Making a power cable for testing of the LED Dome 38
6.4 Refrosting the LED dome 39

7 Discussion 43
7.1 My Work with Legacy Software 44

8 Conclusion 45

Appendices 46

A Figures and Diagrams 47
A.A Diagrams 48
A.B LEDs on the Dome 49

References 49

Chapter 1

Introduction

The work in this project has been carried out as a part of the project NTNU Cyborg. The NTNU Cyborg is a joint project between the Department of Engineering Cybernetics, the Department of Computer Science, and the Department of Neuromedicine and Movement Science at the Norwegian University of Science and Technology. The development of the project is driven by both PhD and MSc students and NTNU researchers. In addition, students from the Cyborg Village in Experts in Teamwork have worked on specific tasks set by either the project in general or by students working on their master's thesis. To grow the neural cells, a Micro-Electrode Array (MEA) is used. This array can both read impulses and send them. There are two different terms when working on the project. The *robot* is used about the hardware that is moving about in Glassgården, while the *cyborg* is the whole project hardware and software which, most importantly, includes the neural cells at St. Olav's Hospital and the connection between them. Though it must be mentioned, that they are used interchangeably.

The work presented in this report will explore the robot at the onset of my work and the possible behaviours to be implemented in the future, what legacy software is and how the theory about legacy software can be used by future students, and continue the work of improving the documentation of the cyborg project. With my work, we hope to bring the cyborg closer to its goal. The work in the specialisation project is in itself a way to further the development of the cyborg, but also a preparation for the work on the master's thesis.

1.1 The NTNU Cyborg Vision

The long term goal of the project is to enable communication between a robot and living nerve cells, to better understand the cognitive processes and consciousness. The electrical output of the neural cell network and simulating environmental feedback, a

closed-loop system is created and the hope is that the network will operate and learn in its environment. With the ability to roam Glassgården and choose its behaviour depending on the current mood, it can become a mascot for both NTNU as a whole, but also the more speci c departments. A mascot can be used on stands, taken to di erent promo events, or roam Glassgården interacting with bypassers and changing behaviour depending on di erent factors and external input. It will be able to navigate around obstacles, recognising people and showing o visualisations of a live feed of the activity of the neural cells through a server. The hope is to peak the technological interests of potential new students and to bring NTNU to the forefront of international research within these disciplines.

1.2 Previous Work with the NTNU Cyborg

There have been several students working on the cyborg during the last years. All have been working on it as part of their master's theses and/or specialisation project. Last year the cyborg was stripped of all unnecessary components to make a base on which the cyborg can be further developed. Other previous work have involved a sel e stick, an iris, arms for the cyborg, face recognition, an arti cial muscle and an Xbox Kinect. A quick explanation of the last years' work is presented below.

2019 - Collected all non-redundant parts of the cyborg into a working base for further development. The motivation was a simpler software structure. Old modules were improved, some kept as they were, and new modules were developed and added such as the behaviour module [5][4].

2018 - Contributed with incremental improvements to the robotics part of the cyborg. The now unused controller system based on behaviour trees was developed. The goal was to represent a more realistic behaviour than the existing system at that time. A program to visualise the behaviour tree while running was also developed. Lastly, a system for object detection and a classi cation system using neural networks was added [17].

2017 - A controller for the robot was made, as earlier modules were able to simultaneously run di erent modules. This controller is the basis for the current controller and is a state machine controller. It used a PAD emotion state model to decide the mood and what to other modules to run. A navigation module using ROSARNL was also implemented [3]. Work was also done to ready the cyborg for presentations. This became the Presentable Robot, and at the same time introduced the idea of a core

on which students could implement their projects. This is where the, at that time called Startup Box now Mode Selector Box, was made [32].

2016 - Work on implementing speech recognition was done this year. This to have the robot communicate with the public. Research was done to examine what the best option for a speech recognition tool would be, and also to convert speech to text. The result was it being able to hold a conversation with informative and humorous answers [14].

(a) 2017 (b) 2018 (c) 2019

Figure 1.1: Previous iterations of the robot.

1.3 Other Ongoing Work

1.3.1 Replacing the Navigation Stack

Lasse Göncz has been working on replacing the navigation stack of the cyborg, as last year MobileRobots closed down [12]. ARNL is the package now being replaced by the ROS navigation stack. Thoughros-arnl still exists MobileEyes and Mapper3 does not and are replaced byrviz and rqt . Thus the cyborg is less dependent on old and dated proprietary code.

1.3.2 Creating a GUI website

Casper Nilsen has created a GUI module to monitor the cyborg. In the future, possible use of this can be to send commands to the cyborg, not just to see its state, position and

behaviour [20]. It is designed for the possibility of adding features. He has also created a guide on how to set it up.

1.4 Work not mentioned any further

- At the beginning of my work with the cyborg, all our computers had to be set up to be able to run the same software as the cyborg. There existed a setup script, but it only included what needed to be installed and not the correct versions. This became a bit of a problem as the commands from the script then installed the latest versions if nothing else was speci ed. A fair amount of work was needed to nd the right versions. One example of this was the installation le of ARIA gave us a version that, when built, gave an error saying a le was missing. After some research on this, we ended up copying the le from the cyborg.
- Starting work on an already established system is a maturation process. As a part of this work was to get to know and understand the cyborg, and the way to do that is to take time as help. It is therefore mentioned here.
- Some time was also used to tidy up the software and code on the cyborg. Anything not in use and therefore redundant such as code left after testing was deleted, though the cyborg was not reinstalled.

Chapter 2

Background

2.1 Hardware

Here, the parts of the hardware of the cyborg which have been relevant for the work in this report are presented.

2.1.1 Cyborg Base - Pinoeer LX

Figure 2.1: The Pioneer LX robot. Image courtesy of [26]

The base of the Cyborg is a mobile robot platform called Pioneer LX from MobileRobots [19][18]. It is a general-purpose indoor platform designed to be able to work around people and is based on the Adept Lynx AIV, also from MobileRobots. It serves as the moving and navigation platform and main computer for the Cyborg on which everything else is mounted. Some software libraries are included which are mentioned further in section 2.2.3. The robot is made for easy integration of custom accessories and sensors and is programmable. Two of the reasons it was chosen are because it can carry up to 60 kg and run both Windows and Linux on the on-board computer. It also features di erent sensors such as a laser to map its surroundings. Other features are:

- Intel D252 64-bit dual-core 1.8GHz CPU and an integrated graphics processing unit.
- 2 GB DDR3 RAM

- Wireless Ethernet connection.
- Front- and rear-facing sonar sensors, and front bumper panel.
- SICK S300 laser scanner.
- Several USB 2.0 ports, a VGA monitor port, 16 In/16 Out digital I/O, and 4 In/4 Out analogue I/O.
- 60 Ah battery, capable of powering the robot for 13 hours.
- Charging and docking station which allows the robot to charge itself.
- Joystick for manual steering and control.

2.1.2 LED-Dome

The LED-dome is an essential part of the Cyborg. Without it, the cyborg would be just another robot driving around and would not be as striking as it is now. In g. 1.1c the LED-dome can be seen on top of the robot body. The dome is made up of two plastic shells, the inner and the outer. On the outside of the inner shell, strips of LEDs of the type WS2812B have been glued in a zig-zag pattern amounting to 791 3 RGB LEDs. Each of them is individually addressable. The byte-array sent to the LED-controller from the dome control ROS node can be accessed the same way as any other array in Python. A graphical overview of the LEDs can be found in g. A.2 in section A.B. The outer shell is made of VIVAK PETG and was vacuum formed along with the inner shell and then frosted using sandpaper. The process of making the LED-dome can be found in [2]. The dome is controlled by a 5V PWM-signal and requires an external power supply of 5V as the LED-controller is not able to produce the su cient power needed.

It is important to di erentiate between the LED-dome, the LED-controller and the ROS node controlling the LED-dome, as it is easy to be confused. In this report LED-dome will be used for the hardware that is the dome, LED-controller will be used for the hardware that is controlling the LED strips, and controller node will be used for the ROS node. Be aware that LED-controller might also be used for the software on the LED-controller as it is not part of the ROS node.

2.1.3 Mode Selector Box

The Mode Selector box started as the Startup Box made by Jørgen Waløen for his master's thesis [32] that ran the scripts corresponding to the choices at start-up. These choices

were ARIA Demo by MobileRobots and launch of the ROS nodes. He wrote a library in C for the Arduino Nano and OLED screen, compiled it with a Make file and uploaded it with avrdude. Last spring the box was updated so it would be able to switch between modes without the need to shut down the whole robot in between. In addition, another mode was added, ARNL. As this change added functionality outside the scope of its original name, the box was renamed Mode Selector box. A description of all changes made in 2019 can be found in [5].

2.1.4 MEA2100 - System Micro-electrode Array

The MEA2100-system is made by Multichannel Systems and is a versatile *in vitro* system made for extracellular recording from microelectrode arrays [29]. It can record cardiac or neuronal cultures, stem cells, or cardiac or brain slices. MEA technology is a powerful tool in electrophysiology research and is based on an idea from the 1970s. The system gives real-time feedback from 60-channels with a 24-bit resolution. Every electrode in the MEA2100-systems is bidirectional, meaning that it is possible to stimulate the cells on each electrode. Neural cells are grown in the MEA system at the neuroscience department at St. Olav's and provides the data used for visualisation on the LED-dome. The system interprets and sends the signals through a computer, which then sends it to the cyborg over a server.

Figure 2.2: Standard electrode numbering of the 60-electrode MEA.

2.1.5 NodeMCU ESP-32S

The NodeMCU ESP-32S is a development board made by NodeMCU [21] based on the ESP-WROOM-32 module by Espessif Systems [28]. The board has a 32-bit double-core architecture with a clock speed of 240 MHz. It has a USB Micro-B connection used for writing to and reading from the board in addition to power. It operates with 3.3V and has 38 I/O pins which support a variety of uses and protocols such as UART, PWM, and output from the on-board DA-converter. Earlier, instead of the NodeMCU ESP-32, the LED-controller was an Arduino Mega 2560 which was replaced in the spring of 2019.

Directly from NodeMCU, the ESP-32 runs Lua TOS real-time system and is programmed in Lua using the Lua IDE. Lua RTOS is designed with three-layers with a Lua interpreter on top, a real-time micro-kernel in the middle, and at the bottom a hardware abstraction layer. The middle layer is powered by FreeRTOS. Installing a minor add-on for the Arduino IDE, allows for programming of the EPS-32 in Arduino IDE with the Arduino programming language [13], changing out the top layer. This allows for much easier usage of the ESP-32 as there is no need to learn another programming language. The Arduino code is only run on one core. Because the ESP-32 has two cores, programming has to be done using FreeRTOS tasks and methods. Otherwise, only one core will be utilized.

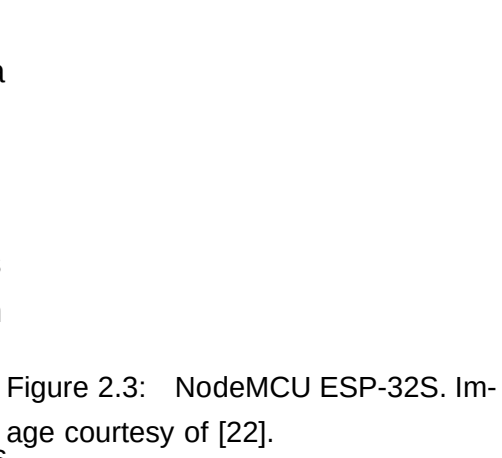


Figure 2.3: NodeMCU ESP-32S. Image courtesy of [22].

Together with the ESP-32, the LED-controller consists of a small circuit with a CMOS-bu er. This is because the LED strips require a 5V PWM-signal to operate while the ESP-32 only provides a 3.3V signal. Therefore a level conversion is needed. The reason a bu er is used and not a passive converter is because a pull-up resistor is to slow to handle the high operating frequency of the LEDs. The type of bu er used is a quad bus bu er of the type 74VHCT125AT. A detailed description of how the software of the LED-controller works can be found in [4].

2.2 Software

All code for the cyborg can be found on the NTNU Cyborg's GitHub page [31]. There each ROS node contains a README explaining the basic details of the node. More in depth documentation can be found in the wiki.

2.2.1 FreeRTOS

FreeRTOS is a real-time operating system made for use in embedded systems. The idea was to design an os which would need very little space in the memory, allowing for fast execution. It is distributed for free under the MIT License [16]. FreeRTOS contains methods for parallel threads, tasks, timers, mutexes, semaphores, and to prioritise threads. For the cyborg, a small install of the NodMCU ESP-32 is needed in the Arduino IDE to program with FreeRTOS. This because the ESP-32 is already running FreeRTOS in the middle layer, as mentioned in section 2.1.5. To give a quick introduction to the methods

used on the LED-controller some examples are given and explained. A further description of FreeRTOS' methods can be found in [11].

```
1  xTaskHandle xHandle;
2  static unsigned char ucParameter;
3
4  xTaskCreate(
5      vTaskLoop,       //Loop-funksjon
6      "TaskNavn",      //Task navn
7      STACK_SIZE,      //Stack str
8      &ucParameter,    //Peker brukt som parameter til task
9      PRIORITY,        //Priotiteten
10     &xHandle,        //Task handle som det returneres til
11     0);              //Core running the task
12
13 void vTaskLoop(void * parameter){
14     for ( ;; ){
15         /* Task kode her */
16     }
17 }
18 vTaskDelete( xHandle );
```

- The first line sees a task created. A task handle is like a reference for the task to be used to delete or create it.
- vTaskLoop is the main loop of the task, and works like the loop()-function on Arduinos. It will run as long as the associated task exists.
- vTaskDelete takes the handle of a task and deletes it.
- xTaskCreate creates the task. If one or several of the parameters taken by the function is not needed, they can be sent as NULL
- The parameter here declared as ucParameter must exist for the entire lifetime of the task. In this case, it is then declared as a static.

FreeRTOS contains several conventions for naming constants, functions, and more. The two most used are the prefixes of the functions xTaskCreate and vTaskLoop in the examples above. The prefix v means that the return type of the function is void, x denotes some result, which is often a handle for a task or a queue [8].

2.2.2 Arduino IDE

The Arduino IDE (integrated development environment) supports coding on a massive number of development boards, not only those made by Arduino. It uses Arduinos own language which is a mix of C and C++. It provides a large number of built-in libraries, as well as any modi ed libraries which have to be open-source. With the installed add-on, the ESP-32 can be programmed with the IDE using C/C++ and FreeRTOS.

2.2.3 Software with the Pioneer LX Base

The Pioneer LX robot comes with several pre-installed and useful tools, some of whom are mentioned here. ARIA is open-source, while some are free, but most must be purchased. Since MobileEyes was discontinued one of the biggest issues have been the accessibility of these tools. Changing this is the work of [12] brie y mentioned in section 1.3.1.

- ARNL - Advanced Robot Navigation and Localisationis built on top of ARIA, made by MobileRobots. It is a set of software packages and used for localisation and navigation, keeping track of where the robot is and allowing the robot to receive a give speci c destination.
- ARIA - Advanced Robot Interface for Applicationscan dynamically control a robot's heading, velocity, and other motion parameters, and can receive operating data sent by the platform. All this is done through its high-level infrastructure or low-level commands.
- MobileSim - Simulator
- Mapper3 - Maps needed in ARNL can be edited and converted to the right format with this tool.
- MobileEyes - Provides a GUI for the robot base. It can remotely control and monitor the Pioneer LX, in addition to create maps. The program can be run from a computer and connect to the robot via the network.

2.3 ROS - Robotic Operating System

ROS is an open-source robotics middleware. It provides services designed like those of an operating system, though is in itself not an operating system. It is a exible framework for working with and developing robot software [24]. ROS works well with modular systems as each node can be removed, added or edited without a ecting the others, and for control

of distributed systems. Nodes of the same robot can be run on different hardware and written in different languages as ROS is language- and platform-independent. All this allows for easy scalability. Their main goal is to provide support and easy reuse of code for research and commercial use.

Since its initial release in 2007, the amount of packages ready for download has been steadily growing. There are now two different versions ROS and ROS2. ROS releases one new release every two years, while ROS2 does so every six months. The lack of real-time support in the first version of ROS has been addressed in ROS2, which also has support for Windows. ROS does not have a core beyond the general structure the same way other operating systems do. Users can configure tools and libraries to fit their application area and robot.

2.3.1 The Concepts of ROS

ROS contains many different methods and concepts. Many of which are used on the cyborg. The most important concepts are presented here.

ROS Master

The ROS Master is an integral part of any ROS system. It can be run with the `roscore` or `roslaunch` commands, which loads the Master along with the other essential parts. The Master provides registration and naming services, as well as keeping track of subscribers and publishers. Its role is to make all information needed by the nodes available. The parameter server is also provided by the Master, but it will not be described in detail here.

Nodes

Nodes are the processes of ROS that perform any computation, connected by the communication streams services, topics or the parameter service to form a graph. They can be compared to atoms connected to form molecules, and are the part that makes ROS so modular. They can easily be restarted in the case of a crash and can be switched out just as easily. Nodes communicate peer-to-peer through the communication streams and can identify each other by name the same way a Unix system uses a file path to locate a file. ROS nodes are part of a package. Most often each package contains only one node, usually with the same name as the package, but that is not a must. A package can contain as many nodes as needed.

Topics

Topics are the channels nodes use to send messages over to communicate. Topics are defined by name, just like messages. They are asynchronous and follow a publish-subscribe model, where there is no limit to the number of topics a node can subscribe and publish to. A node can publish on a topic it is also subscribing to. Nodes are not aware of the other nodes they are communicating with, or if anyone has received the messages sent. To receive data from another node, nodes subscribe to the relevant topic. Using the command `rostopic` in the terminal yields information about which topics exist, which nodes are subscribing and publishing to them, and what type of message types are used. Because topics are asynchronous, if synchronous messages are needed, it is often better to use services. In Python, a publisher and subscriber are created by writing:

```
1 bool_pub = rospy.Publisher("topic_name", bool, queue_size = 10)
2 rospy.Subscriber("topic_name", bool, callback_function)
3
4 bool_pub.publish(message)
```

The first line declares that the node is publishing messages of the type `bool` on the topic named `topic_name`. `Queue_size` is used to limit the amount of messages in case a subscriber cannot receive them fast enough. The second line declares a subscriber to the same topic. Every time a message is received on the topic, the function `callback_function` is called and takes the received message as an argument. The last line is used to publish a new message to the topic.

Messages

Messages are naturally enough, messages, used to communicate between the nodes. They are sent over topics by a `publisher` in a node and received by a `subscriber`. A message contains simple data structures such as string, int, bool, float, and arrays of primitive types. They can also support arbitrarily nested structures and arrays. Structs in C are quite similar to messages. Messages can also be used to send fields of data and constants to help interpret the data field. Two or more nodes sending and receiving the same message have to use a defined name of the message type. Otherwise, they will think they are using two different message types. This will also mean that one or both will not be able to publish or subscribe to a topic, as they are defined with a specific message type to be used. A typical message can look something like this:

```
1 Header header
2 int8 CHARGING_UNKNOWN = -1
3 int8 CHARGING_NOT = 0
4 int8 CHARGING_BULK = 1
```

```
int8 CHARGING_OVERCHARGE = 2
int8 CHARGING_FLOAT = 3
int8 CHARGING_BALANCE = 4

bool[] array_of_bools
int charging_state
float32 charging_percent
```

Initialising a message can be done by writing:

```
message = MessageType()
message.data = value
```

Where the first line initialises a message of the type MessageType() and the second line assigns a value to the argument data. In ROS there are many standard message types in the library std_msgs. When defining a message type, they are stored in .msg files in a subfolder called msg/ in one of the packages using the message type. When doing this, the message files have to be added to the CMakeLists.txt file for that specific package. This so they can be translated to source code.

2.4 SMACH - State MACHine

SMACH is a task-level architecture for creating complex robot behaviour. It is an open-source Python library and can be used to construct concurrent and hierarchical state machines. SMACH is useful when you want a robot to execute a complex plan of which all states and transitions can be explicitly described [27]. It allows for fast prototyping, complex state machines and introspection. If your system has unstructured tasks, is low-level or, as they state on their documentation page, when you want to smash something, then SMACH is not the best option. SMACH is not dependent on ROS and can be used separately, but the executive_smach stack allows for integration with ROS. It also includes integration of the actionlib package, which is a part of ROS.

SMACH has two main interfaces, Containers and States. Containers are a collection of one or more states and are the implementation of execution policy. The containers have a flat database to pass data between and coordinate states. The states in a container are stored like a dictionary. The most important job of the container is to define the transitions between states, and also what to do if a state wants to preempt another. States can mean different things depending on the context. In SMACH, a state corresponds to a state of execution of some task and all potential outcomes of that execution. The states are different from traditional state machines in that they describe what the system is doing locally, not the configuration of transitions between states. A SMACH state

machine can be nested, meaning that one state machine can be a state in itself.

2.4.1 Creating a SMACH State Machine

The following is an example of how to implement a SMACH state machine with two states:

```
1  sm = smach.StateMachine(outcome = ["outcome3", "outcome4"])
2  sm.userdata.sm_variable = 0
3
4  with sm:
5     smach.StateMachine.add("State1", State1Func(some_function, arg1),
6              transitions = { "outcome1":"State2",
7                              "outcome2":"outcome3"},
8              remapping   = { "state1input":"sm_variable",
9                              "state1output":"sm_variable"})
10
11    smach.StateMachine.add("State2", State2Func(),
12             transitions = { "outcome1":"State1"},
13             remapping   = { "state2input":"sm_variable",
14                             "state2output":"sm_variable"})
15
16 outcome = sm.execute()
```

The first line declares a state machine with the name sm and its outcomes outcome3 and outcome4. The container is made on the fourth line, and adds the states State1 and State2 on lines 5 and 11, respectively. Transitions are naturally enough the transitions from one outcome to another outcome or state. Remapping is not necessary but a good practice. It makes it easier to follow the flow of the code and not confuse variables used in states and state machines. To implement State1 from the previous example you inherit the state.State base class:

```
1  class State1Func(smach.State):
2      def __init__(self, function_inputted, arg):
3          #state initialization
4          smach.State.__init__(self,
5                   outcomes = ["State2", "outcome3"],
6                   input_keys = ["state1input"],
7                   output_keys = ["state1output"] )
8          self.var = arg
9          self.func = function_inputted
10
11     def execute(self, userdata):
12         #state execution
13         if userdata.input == 1:
```

```
14              return "outcome1"
15          else:
16              return "outcome2"
```

The class is initialised in the first __init__ function and the state is initialised in the second. There the outcomes, input keys and output keys are set. The initialisation functions must not block further execution. Any other variables the state uses are set after the second, but still in the first initialisation function. The execute function is where the behaviour of the state is implemented. Execute can block for as long as needed. When returning from this function the current state finishes executing.

2.5 Cyborg Software Structure

The cyborg consists of many modules, and from experience, I can say that it might be hard to get a grasp of the overwhelming amount of code the first time seeing it. Here is therefore a quick explanation of each node. Figure 2.4 is a simplified class diagram of the whole cyborg, with published and subscribed to topics. Figure 2.5 shows the relation between ROS nodes and topics. Both can also be found on the NTNU wiki [30].

Audio - A ROS node in charge of playing audio files, and also contains text-to-speech functionality.

Behaviour - A ROS node that adds and executes behavioural presets. It exploits the emotional state of the cyborg and provides emotional feedback. New behavioural configurations are added in the behavior.launch file. Also decides what future states to execute.

Event Scheduler - A ROS node which provides functionality related to publishing scheduled events. It also monitors other system events like low battery.

Primary States - Gathers action server states too complex for the behaviour module, and states that do not produce outputs. Can execute state changes, and provides emotional feedback to the controller. A ROS node.

Command - A ROS node, implemented, but not in use at the moment. It is a command module run on an external computer to monitor the cyborg.

Controller - The main state machine on the cyborg which manages all other nodes. Organises all actions in the state machine, handles the emotional system and motivator. Also a ROS node.

Navigation - A ROS node which handles all high-level navigational execution on the cyborg through the action server.

Mode Selector - Not a ROS node, but contains the code used on the external box used to choose which mode the cyborg should run.

LED Dome - Controls the behaviour of the LED-dome and sends output to the LED-controller to set the LED strips. It does not choose the di erent visualisation modes, but receives them on the topic/cyborg_visual/domecontrol from the behaviour module.

ROS Arnl - A ROS node developed by MobileRobots which handles communication between the robots sensors, actuators and navigation library, and the rest of the ROS nodes.

ROSARIA - A ROS node and provides a ROS interface for most MobileRobots bases such as the Pinoeer LX.

Figure 2.4: A simplified class diagram of the whole cyborg. From the NTNU wiki [30].

Figure 2.5: Overview of the different ROS topics and nodes on the cyborg. Oval modules are nodes and the rectangles are topics. Outgoing arrows from the nodes are publishers and incoming are subscribers.

Chapter 3

Legacy Software

3.1 The De nition of Legacy Software

I want to start this chapter by looking at what legacy code is. Legacy code is often thought of as either just old code, something you have not written yourself and inherited with a system or the code of a system you have bought. The last one can easily be mistaken for proprietary software code, which is actually non-free software that someone holds the property rights too. Also known as closed-source code. Some de ne legacy code as code you have gotten from someone else, often someone who no longer works on the project. For example, you start in a new job and you get some software from someone else in the company, or your company has gotten the software from another company. It is somebody else's code, that you must change. I will mostly base this chapter on the methods presented in Michael Feathers' book, and it is a recommended reading for anyone working on larger software projects. There seems to be a consensus among developers that Jonathan Boccara and Feathers books are the leading books on this subject.

3.2 Approaching Legacy Code

When I think of the word legacy code, I also think of code inherited from somebody, often a mess of hard to understand and tangled spaghetti code. In the computing industry, it is often used for code that is di cult to change, like when a feature cannot be added without breaking another logic. Wikipedia describes it as code that is no longer supported or manufactured [15]. In his book "Working E ectively with Legacy Code", Michael Feathers states that to him "legacy code is just code without tests" ([9], p. xvi). His opinion is that you can have a clean and well-written code, but with the tests, you can change your code and then verify that it works much quicker. Jonathan Boccara

presents de nes legacy code by three qualities in his book "The Legacy Code Programmer's Toolbox" [7]. Firstly it is hard to understand, secondly you are not comfortable changing it, and thirdly it is code you are somehow concerned with.

Boccara states that there are two mindsets when approaching legacy code, the natural mindset and the e ective mindset. Most often it is the e ective mindset that works best, while the natural mindset tends to get stuck. Being annoyed by poorly written code is a primal reaction but being a software developer, your job is to be rational. The code you are annoyed by is still code you have to work with and is the reason the application is the way it is. For better or worse. Complaining will not make it better, and it creates a negative mindset. Like for many things in life, do not complain about something you do not intend to improve. The legacy code you have been handed is now your code, and taking ownership of it is not the same as taking the blame for whatever thing you disagree with.

3.2.1 Unfamiliar Code

Familiarising oneself with unfamiliar code, especially legacy code is daunting. Just reading code can make you dizzy and when the documentation poor it is not helping. There are many ways of gaining understanding, but they take e ort and most people are just looking for the most immediate way to understand.

One technique is to start drawing. Take notes and draw the connections between each module and its important features. They are just sketches and do not have to be full-blown UML, although they do make a basis from which it is easier to create UML diagrams. It can also help to redraw your rst notes into something more coherent. There is no guidance for how to draw code, and while it may be tempting to use proper UML syntax, just using something you can understand later is key. Another technique is listing markup. It is especially useful with long methods. The way it is used depends on the type of code you work with. The four main activities of listing markup are separating responsibilities, understanding method structure, extracting methods, and understanding the e ects of a change. When separating responsibilities it is often smart to use colour markers to group things, and if some belong together you can indicate that by putting a special symbol next to them. To help understand the structure or e ects of a change marking the methods in some way is helpful.

One of the most helpful techniques is *scratch refactoring*. Just start moving things around. Reorganise it in a way you understand. Do not go back and check things after taking it out. It is a great way of stripping the system down to its essence. There is one problem though. If you do not have any tests and you make a mistake when refactoring,

it can lead to you having a false understanding of the system. Lastly, if there is any unused code. Delete it. It will only get in the way.

3.3 The Mechanics of Change

The reasons for changing software in the first place can be many and varied, but there are four main reasons.

- Adding a feature
- Fixing a bug
- Improving the design
- Optimising resource usage.

Features are what users depend on and the behaviour they provide is the most important thing about software. Bugs can be errors in the code, code that does not behave correctly, or they can be more blurred. An example can be when a client wants a button moved from right to left. Is that a bug or adding and removing features? Depending on the perspective it can be both. When improving design, we use a method called refactoring. Instead of rewriting chunks of code or doing code cleanup, we are making small, incremental changes supported by tests. Lastly, optimising is closely related to refactoring. But the reason being improving the usage of resources, like faster memory access or less mobile data usage.

There are two primary ways to change software according to Feathers: Edit and Pray or Cover and Modify. Edit and Pray is what most people would naturally do. They would make sure to understand the code, plan their changes and implement them. Then they test and poke around to make sure nothing else was affected. Cover and Modify uses a safety net of tests like a cover over the existing code to make sure no changes we do affects the code negatively.

One person rarely works alone on a big project, and different people write code differently. Teams will then rarely have clean code making the tests even more important. Adding a test is a way of changing the code, and when changing the code we need to verify that those changes did not negatively affect existing behaviour, to that we need tests. A little dilemma right there and Feathers calls this the Legacy Code Dilemma. If there already was good test coverage in the code and continuous integration and updating. Chances are it is not that big of a deal, but without them, it gets worse.

So what is testing? It is the process of executing some code with the purpose of finding errors and bugs. There are several different types of tests like unit tests, performance tests, integration tests and acceptance tests. To mention a few. For work with legacy code, unit tests are the most important. They test small, localised parts of the code and gives instant feedback. If unit tests do not run fast, they are not proper unit tests. Feathers proposes the Legacy Code Change Algorithm:

- Identify change points
- Find test points
- Break dependencies
- Write tests
- Make changes and refactor

One method to hide legacy problems is to use APIs. Using them the architecture becomes such that you can switch out what happens when the APIs are called without worrying about the "outside". This is an old method in software development, but the important aspect is to organise and maintain what happens when they are called. To continually update them. The Strangler Pattern describes this gradual change well. Over time, the original legacy service disappears and the new code takes over [10].

One more important concept presented by Feathers is seams. Which are defined as "a place where you can alter behaviour in your program without editing in that place" ([9], p. 31). Seams enable you to separate pieces of code under testing but also enables you to check and verify the behaviour of the code. There are methods and tricks to make the code testable, some more obvious than others. To make code unit testable you need to cut dependencies and introduce barriers. The two main reasons being sensing and separation. Sensing by executing a piece of code and check and verify its effects, and separation to isolate the specific parts of the code to be able to test it. Breaking dependencies can be hard to do safely.

3.4 Changing Software

Changing software can be tricky. Especially if you are short on time. Generally, there is a loss of quality when quantity is prioritised. Feathers opinion is that writing test might feel slow in the beginning, but ultimately you will save time by not having to fix bugs and debugging becoming more convenient.

3.4.1 Changing the Code Quickly

When you need to make changes but cannot afford or do not have the time to break dependencies or get tests in place, there are still some useful techniques. They have to be used carefully because you add tested code, and unless you cover existing code you are not testing its use. The first technique is called *Sprout Method* and can be used when a new feature has to be added and is all new code. It is easy to include a test when writing this new code and should be used when the new code can be seen a distinct piece of work. The next technique is *Sprout Class*. It is useful when objects cannot be created because of some tangled dependency. The complexity of implementing new code with the Sprout Class is its main disadvantage but requires less invasive changes. The third technique is called *Wrap Method*. It can add behaviour to an existing method, without increasing the size of it. The last technique is the *Wrap Class* and is the class level version of the Wrap Method. It adds new behaviour to an existing class instead of a method by using Extract Implementer and Extract Interface. This technique of creating objects of a class that wraps another class is called the *Decorator Pattern*.

3.4.2 How to Add a Feature

The most powerful technique on how to add features is *Test-Driven Development* (TDD). You write a test case that is sure to fail for a method that will solve some problem. If you can write a test case for this nonexistent method, you have cemented your understanding of what this nonexistent method should do. The *Test-Driven Development Algorithm* is defined as:

1. Write a failing test case
2. Get it to compile
3. Make it pass
4. Remove duplication
5. Repeat

When using the TDD algorithm for legacy code, we can extend it by adding an item before number one: get the class you want to change under test. The key with TDD is that it allows you to focus on either refactoring or writing code.

TDD is not the only technique for adding features. With object-oriented programming there is an option to use inheritance to introduce features without modifying the class directly. After the feature is added it can be integrated. The key technique for doing

this is called Programming by Difference. It is used much in the 80s but it became less popular when developers noticed that inheritance can be fairly troublesome if overused. Programming by difference lets you introduce variations quickly, and then capture the new behaviour.

3.5 Dependency-Breaking Techniques

There are many different techniques for breaking dependencies in software, and this is not an exhaustive list at all. All mentioned below preserves behaviour, and are technically refactorings used to get tests in place.

Adapt Parameter - transforms an external interface into an internal interface, and often used when you for some reason cannot use Extract Interface on a parameter's class or when the parameter is hard to fake. Be aware that adapt parameter does not preserve signatures. It can be risky to make invasive changes without tests in place.

Break Out Method Object - the idea is to move long methods into new classes. Objects are created using that new class and are called method objects. After using this method it is often easier to write tests for the new class than the old.

Definition Completion - when your programming language has the ability to declare a type one place and define it in another, it can be used to break dependencies. Usually, this method is only recommended for the worst dependency situations.

Encapsulate Global References - tries to encapsulate all globals that are used or modified near each other into the same class. If you find that the new class name is already in use, consider renaming one of the classes. Start with data or small methods, and move to larger methods once tests are implemented.

Expose Static Model - similar to Break Out Method Object, but used when the method you are working with is smaller and does not use instance data. If a method does not use any instance data, make it static until you figure out what class it belongs to.

Extract and Override Call - an ideal method to break dependencies on static methods and global variables. Useful when there are a limited number of calls against a global. It a refactoring method.

Extract and Override Factory Method - also a refactoring method, used to get around classes that are hard to test because they initialise lots of objects in their constructor. This method is not possible to use in C++ as C++ does not allow virtual function calls to resolve functions in derived classes.

Extract and Override Getter - introduces a get function for the instance variable that you want to replace with a fake object. Often used when the Extract and Override Factory Method cannot be used. Be aware that it is important to delete the testing instance, and to be conscious of object lifetime issues.

Extract Implementer - turns a class into an interface by drawing it out and putting the contents of the class into a new implementation class.

Extract Interface - you create an interface for a class and the declarations of all the methods. Then implement the interface to separate or sense, and to test you pass a fake object to the class. It is considered one of the safest techniques, and often the best choice when breaking parameter dependencies. When you cannot extract an interface from another interface, you will have trouble using this method.

Chapter 4

Reevaluating the Cyborg

At the beginning of my work with the cyborg, I had to familiarise myself with ve years worth of work. This is not a trivial task as the system is complex and includes several new tools. Last year an e ort was made to make the robot more coherent and leaner. All redundant code and software were removed, leaving only that needed to create a basis on which to build the cyborg. The master's thesis The Cyborg v3.0[5] collected information about most of the modules, and improvements needed to nalise the cyborg. I agree wholeheartedly with the statement that 'every robust system needs proper documentation in order for it to be easily understood and properly maintained.' ([5], p. 27). This chapter aims to review the status of the cyborg at the onset of this project, point out issues that need improving, and suggest directions to take the behaviour module in. Both with regards to the documentation and further development of the software. It seeks to carry on the work of making a robust platform on which the cyborg can continue to grow. This platform has now been started but minor additions, as well as bigger changes, are needed to nalise it. Hopefully, the cyborg project will continue for many years, thus needing in-depth and organised documentation.

4.1 Previously Proposed Tasks

Former students have proposed several tasks to be done to satisfy the goals of the cyborg project. They are presented below with comments on their status.

Recovery Behaviour - where the cyborg would be able to restart itself if it gets stuck, loses track of its location, or similar. All without human intervention. A part of this was suggested to be to publish a message to the new GUI website where noti cations could be enabled so that someone could come and help, should it ever need human help.

Docking Behaviour - the Pioneer LX robot is delivered with automatic docking which is used by the navigation module, but the behaviour when it is there is still missing. When to dock and leave the dock is not implemented. A new docking behaviour has to be implemented with the replacement of the navigation stack.

Emotion Con guration Guide - the emotional feedback for behaviours is now chosen by the coder before running. A set of tuning guidelines detailing how to set emotions in the controller or an implemented function for the cyborg to set these by itself is needed. This is related to an upgrade in the state machine.

Change Navigation Stack - because of the issues with Mobile Robots closing down changing the navigation stack will remove all proprietary code. This has been the work of Lasse Göncz in his specialisation project [12].

New Visualisations - as the cyborg roams Glassgården it will become repetitive quickly if there is not a great array of animations to choose from. Work on this has been started and two new implementations can be seen in chapter 5.

More Advanced State Machine - the current state machine works, but for similar reasons more visualisations are needed, so is the state machine in need of upgrading.

Change to ROS2 - ROS2 provides software for object detection which can not be found in ROS. As this is not in use at the moment, upgrading ROS has not been a priority. Though if it is to be used as the system is now, a bridge between ROS and ROS2 has to be used. Migrating entirely to ROS2 would mean an overhaul of the rest of the cyborg as well. The reasons being that ROS2 does not work with Python 2.7 used now, among other things. The operating system would most likely need updating too.

Integrated Object Detection - object detection has been implemented previously, but is not in use at the moment. All hardware needed for object detection has already been acquired.

Computer Upgrade - the cyborg sometimes struggles with handling di erent tasks such as navigation and visualisation of neural data, and will need an upgrade sometime. Previously, when using object detection was implemented, a Jetson Tegra TX1 was used to run that module. Unfortunately, running two computers at the same time on the same system presents some complications. It would be preferred to use only one, and the current computer is not capable of running things such as object detection.

Refrost the LED-dome - the LED-dome disperses the LEDs to some degree but the strips are still visible, which is undesirable. This has been done and is presented in section 6.4.

Update Text-to-Speech Functionality - the text-to-speech module used now is a Python library speech synthesizer called espeak It works but sounds horrible. A recommended replacement is either IMB's Watson text-to-speech or Google Cloud text-to-speech. Both are online and works a great deal better than the one currently in use. An online text-to-speech engine would also mean less processing power needed from the robot.

Create a GUI - the exists a very simple node that monitors the cyborg's state, its previous state, emotional state, and more. This has not been used much and had to be run on a separate computer while the cyborg was running. To better monitor the cyborg and its behaviour, a new GUI is needed. Casper Nilsen has been working in implementing a new GUI in his specialisation project.

4.2 Discussion of Proposed Tasks

Many of the tasks mentioned above have been started or are already scheduled to be implemented the next semester. It is worth mentioning that since the development of the software on the cyborg is done mainly by students, the progress is not always smooth, and di erent students may have di ering opinions of the methods to use. Thus the tasks being focused on also di er.

Lasse is working on the replacement of the navigation stack, and with that done, implementation of a recovery and docking behaviour will be possible to implement. It will need implementation in more than just the navigation, because the states to execute these behaviours are still missing, meaning they need to be added to the controller module as well. Casper is developing a GUI remote control, which will have many di erent possibilities. It may be used as a remote control, either directly or indirectly. It will be able to show the cyborgs state and position. Another possibility is a live camera feed from the camera used for object detection. If there is an event in Glassgården, a project manager may be able to use the website to tell the cyborg not to roam around at di erent times during that day, so to not interfere with the event and steal the spotlight.

After the cyborg was stripped of all redundant and unused software, the structure of the software was changed, and the documentation was also updated. Though all this

information was not stored in a database of some sort to make it even easier to access. Because of this, we all agreed that collecting all software in one GitHub repository and the documentation in the repository's wiki page was necessary. The documentation will need updating every time there a change is done, which is the whole point. Instead of having lots of reports to read to find the information you need, you go to the specific page in the wiki. This wiki should also include what different parts are and do, and how to change them. In addition to the proposed GitHub wiki, there already exists a more general wiki [30]. My opinion is that this has not been kept properly up to date, though some parts are more recently updated than others. There are diagrams used in several of the reports and masters theses that are very useful for the system as it is now. I would have liked for them to have been added to the general wiki as well, as they illustrate the layout of the whole system and more specific parts. Diagrams are an important tool for graphically present the connections in a system.

4.2.1 The Cyborg in General

The cyborg in general is far on its way to becoming the mascot it is supposed to be, even though there is still a way to go. To aid in the setup of students' working computer the setup script needs to be updated. As I mentioned in the introduction chapter, a fair amount of work was needed to get the right setup. Because of this, it should be updated to include the right versions of everything that needs to be installed. These dependencies should also be included in the documentation for each node and module. After several years of work by different students, the robot has not been reinstalled many times. Especially after MobileRobots closed down, in fear of losing software on it. Even though the software structure has been reorganised and redundant code removed, there were many old versions of the code outside of the ROS workspace, many used for testing. A quick cleanup was needed, freeing up space on the robot as well.

The Mode Selector Box stopped working during the semester, and it has not been properly looked at. The suspicion is that the hardware has failed in some way since the software has neither been changed or re-uploaded. Hardware-wise it is in need of an upgrade regardless, as most parts are kept in place by glue.

Computer vision and object detection have been implemented previously, and while it is not in use now the applications are many. The main though behind the use of computer vision is to avoid objects in combination with the LIDAR already in use, and to interact with people. The cyborg tends to get stuck because of the joints between the tiles in the floor, and object detection can be used to improve this by telling the cyborg that there is indeed nothing in front of it.

4.2.2 The State Machine

As the state machine is now, it organises and coordinates all ROS nodes. Actions from ROS' actionlib servers are organised into states. The state machine features an emotion system, motivator and a database. The emotion system is based on a PAD emotion state model and handles the cyborg's mood. The motivator decides what other modules to run when scheduled or external events are unavailable. Even though the controller works, it changes between behaviours very quickly. Too fast to do much. In 2018 a new way to implement a control system was proposed and uses behaviour trees [17]. Either in a combination with a conventional state machine or alone. Behaviour trees are mathematical models of execution plans often used in game development. Instead of the building blocks being states, they are simple tasks. Those simple tasks are then used to construct very complex tasks by combining them in a tree fashion. Another key element with behaviour trees is its ease of human understanding making them less prone to errors. Another possibility is to use a cognitive architecture or an AI. A system using an AI of some sort could be able to learn new behaviours, have better emotional intelligence, and when to be most active during the day depending on the public's activity. This would require more computational power, but also someone with more experience in that eld to implement. But this is not something to prioritise at the moment. The main reason to upgrade the state machine is to have a more natural and realistic behaviour of the cyborg. How the mentioned methods of implementing the state machine would be in uenced by the work with the neural cells and their continuous connection to the robot has not been investigated.

4.2.3 Behaviour

The cyborg can potentially do a lot of fascinating and cool things when roaming Glassgården. At the beginning of this project, it had three di erent animations in addition to the visualisation of MEA data. This was combined with movement and sound e ects. Most often the movement was just wandering around, not going to a place in particular.

Since the cyborg is supposed to move about, it will also need navigation as part of its behaviours. It may be to move to a particular position or to move in a preset pattern. These will need to be investigated and programmed once the new navigation stack is complete. There is also a need for more animations for the LED-dome to accompany new behaviours. The possibilities are almost endless, but we run the risk of them being very complex at the moment as there is no mapping function for the LEDs on the dome.

We could also borrow some ideas from Star Wars in the form of R2-D2 and BB-8. These droids do not talk back to people when asked about something or talked to in

general. Even though some people seen to understand them, to understand the general idea of their responses only beeps are needed. This is one direction to take the cyborg, instead of having something more similar to Amazon's Alexa or Apple's Siri.

An idea to improve the behaviour of the cyborg is to implement a return event to be sent upon completion of a behaviour preset execution. This would be needed anyway if the cyborg is to navigate to a speci c position in Glassgården as part of a behaviour. A return-event would remove the need for timers or counters used to change behaviours and would enable the cyborg to do something else before the motivation module decides it is bored of doing the same thing. But it would also complicate the state machine. Previously it has been suggested to divide the behaviour presets into several di erent launch les, as the current has grown quite big and will continue to grow as more possibilities of behaviour are added.

If the state machine is changed the behaviour module might need to change too. More information about the use of behaviour trees in ROS can be found in [25] and [6].

Right now we have no feedback to the neural cells of the cyborg. Though it is implemented on their side of the connection, the cyborg sends no feedback. Nor is there an implemented live connection to the cells. When shown MEA data on the robot it is done from a prerecorded le, but the possibility to send both ways are there. The MEA data is not used to a ect the emotion and behaviour at the moment, but that is one other use of the data in addition to visualisation of it. The data would then need some kind of processing to extract features before it is used to set emotions or choose behaviours. As an interesting side note, the cyborg's emotions could be visualised on the LED-dome using di erent colours to visualise each emotion. Visualising the emotions is not dependent on the MEA data, and could be implemented today with the emotions already in place. The present emotions are not speci ed as happy, angry, sad, or similar, but rather using the PAD emotion model where three numerical dimensions Pleasure, Arousal, and Dominance are used to represent all emotions. There are many possibilities when deciding the direction to go with the behaviour.

Chapter 5

Visualisations

There is a need for more animations for the cyborg. To give it more options to choose from when driving around. After studying the existing code I have made two more, as seen in section 5.1 and section 5.2 further down. It took some trial and error since as of now there is no mapping-function for the LEDs on the LED-dome. To find the exact LEDs needed for a specific visualisation, I found there are two ways to approach this: either choose some random indexes of the LEDs and then see what happens, or count forward to those needed. When coding, I chose a combination. Choosing approximately how something would look, and where each light needed to be, and then counting the rows and columns roughly, eventually fine-tuning the indexes. For easier guessing and counting I felt the need to make a graphical visualisation of each LED's index, which can be seen in fig. A.2.

An important issue to keep in mind while making visualisations is that the LED strips need a fair bit of power. If all 791 would be on at full power most of the time, the battery would be drained considerably fast. Thus they cannot be used at full power, something that should not be necessary either, as 791 LEDs at full power equals something similar to a massive flashlight. There have also been done some minor adjustments to some of the existing visualisations. Among them changing the siren animation so that the stripe where the halfway point of the LEDs is is not in the middle, but rather at the edge where two stripes meet. Small changes like this give a more complete and higher quality feel, even though it is not something consciously thought of when seeing it. The eyes animation has also been altered. Previously the eyes were looking straight up towards the ceiling. Now they have been moved a bit so that they look more forward than up.

5.1 Battery Visualisation

This animation will go along with a coming behaviour where the cyborg navigates to its charging station or is placed there, and the ensuing charging is then visualised using this animation. It may also be used to show when the cyborg has a low battery percentage. The reason for this is to still make the cyborg somewhat interesting even though it is just charging. The small battery drainage in icted, and thus the extra time needed, by the LEDs being on has been deemed so low that it is negligible.

The le added to neural_interpreter is called charge.py, and it subscribes to the ROS topic rosarnl_node/battery_status published by the ROS node rosarnl. The reason the battery charge percentage is not passed as an argument to the function is that domecontrol() reset the value after passing it to charge(). Meaning the next time it passed the battery percentage it would be zero. The LEDs are updated more often than the callback function received a new value to set, the result being that they blinked so quickly it was impossible to discern. The array is also ushed each iteration so that it is possible to show the battery declining. The code is explained further down. The LED-dome is lled with green as the battery is charged, or emptied as it is drained. To further get across to people watching what this visualisation is, there is a small red battery at the top. All those green LEDs have to be disengaged, because the red blends with the green making it hard to see. Unfortunately, red-green colourblind people will not be able to appreciate this visualisation.

Figure 5.1: Visualization of the robot charging. Here the charge is around 80 percent.

```
1  import system.settings as settings
2  import rospy
3  import math
4  from rosarnl.msg import BatteryStatus
5
6  class Charge:
7      def __init__(self):
8          self.rate = rospy.Rate(3)
9          self.isStatic = False
10         self.battery_charge = 0.0
11         self.leds_charged = 0
12         self.red_battery = [245, 246, 247, 281, 280, 279, 278, 277,
    313, 317, 350, 346, 381, 385, 418, 414, 449, 450, 451, 452, 453]
13
```

```
14        subscriber_battery_status = rospy.Subscriber("/rosarnl_node/
    battery_status", BatteryStatus, self.battery_status_callback)
15
16    def battery_status_callback(self, message):
17        self.battery_charge = message.charge_percent
18
19    def render(self, input_data, output_data):
20        if not settings.CHANGE_REQUESTED:
21            for i in range((settings.LEDS_TOTAL*3)-1):
22                output_data[i] = 0
23
24            self.leds_charged = int((self.battery_charge * (settings.
    LEDS_TOTAL/100.0))//1)
25            for leds in range(self.leds_charged):
26                output_data[leds*3+1] = 10
27
28            for x in self.red_battery:
29                output_data[x*3+1] = 0
30                output_data[x*3] = 10
31            self.rate.sleep()
```

- The imported files math and rospy are built in libraries in Python and ROS, respectively. System.settings contains definitions for the LED-dome and BatteryStatus is the message type sent from rosarnl on the topic battery_status. The battery charge value is a float32.
- The function __init__() on line 7 sets the values for the class. Rospy.Rate() is used to keep a specific rate for a loop-fuction, and takes an int argument in Hz. The Self.isStatic variable is checked by the execute-function on the state machine to determine if the animation needs updating while running. The red_battery variable contains the indexes of the LEDs that make up the small battery.
- Line 14 makes the subscriber to the rosarnl_node/battery_status topic. Every time a message is received the callback function will run.
- The callback function is declared on line 16 and sets the variable battery_charge to the new battery percentage received.
- Render() checks if the state machine has ordered the state to change. If false, the whole output array is reset and then it calculates the number of green LEDs needed and iterates through setting the correct amount. The output array contains 791 3 values, where every third is a red. Every visualisation is required to call its main function render by the implementation of domecontrol.py.

- The last loop iterates throughred_battery and resets all indexes in the array to 0 to turn o the green and then to 10 to turn on red.

5.2 Roadwork Visualisation

After making the rst visualisation the understanding of the system and its context grew, making it easier to implement another. An animation previously suggested be made was a roadwork animation. To get an orange light from the LEDs, we need a combination of several. An interesting fact when working with light and colours is that the reason red, green and blue is because they are additive colours. This means that they are emitting light, and the more you combine the lighter the accumulated colour becomes. This contrasts to subtractive colours, like those used when painting, where the bases are red, yellow and blue, and when combined they become darker. Because there is no orange LED on the strips, a quick Google search resulted in the code RGB(255,140,0) for a deep orange.

Figure 5.2: The roadwork visualisation without the outer dome of the LED-dome.

Unfortunately, this code turned out to be too yellow, in addition to the light being too powerful. Hence I started by cutting all values by half and experimented with combinations until satis ed with the colour code RGB(150,40,0). This is fairly similar to the colour found on building sites and road construction vehicles.

```
1  class Roadwork:
2      def __init__(self):
3          self.rate = rospy.Rate(3)
4          self.isStatic = False
5          self.previous_light = False
6
7      def render(self, input_data, output_data):
8          if not settings.CHANGE_REQUESTED:
9              if self.previous_light:
10                 for i in range((settings.LEDS_TOTAL/2)+5):
11                     output_data[i*3 ]   = 0
12                     output_data[i*3+1]  = 0
```

```
13                for i in range(settings.LEDS_TOTAL/2+5,settings.
    LEDS_TOTAL):
14                    output_data[i*3 ]    = 150
15                    output_data[i*3+1]   = 40
16
17                self.previous_light = False
18
19            else:
20                for i in range((settings.LEDS_TOTAL/2)+5):
21                    output_data[i*3 ]    = 150
22                    output_data[i*3+1]   = 40
23                for i in range(settings.LEDS_TOTAL/2+5,settings.
    LEDS_TOTAL):
24                    output_data[i*3 ]    = 0
25                    output_data[i*3+1]   = 0
26
27                self.previous_light = True
28
29            self.rate.sleep()
```

- The same way the battery animation sets the variables, they are set in the init -function. Previous_light is used to turn on and o the LEDs, every other time on the halves of the dome. The initial value of this variable hardly matters as the opposite value means the opposite half of the dome is lit up rst.
- Render also works the same way as any other render-function, only the content is di erent.
- Line 9 checks previous_light and depending on the value di erent for-loops are run. The only di erence being which half of the LEDs are turned on and o . The reason for the +5 is so that a strip is not cut in two, as mentioned before.

Chapter 6

Minor Additions to the Cyborg

There were several improvements, both minor and more comprehensive, done this semester. Some of those are collected in this chapter and presented below.

6.1 Udev Rules for Accessing USB Connections

Previously there existed a function for the Arduino Mega named find_arduino_port which did not work when the LED-controller was changed to the ESP-32. The path was then hard-coded, and if a device was unplugged it was a nuisance to be sure you had the correct port. Because of this, I decided to implement some udev rules for the cyborg. Udev rules are flexible and powerful and can be used to rename a device to whatever is needed from the default. They can change permissions and ownership of device nodes, rename network interfaces, launch a script when a device node is deleted or created, among many other things. The use, in this case, became to rename the path to a specific device to a predetermined name based on the info a computer reads of a device. On the wiki page on NTNU Cyborg's GitHub is an in-depth explanation of how udev rules are made and used.

The LED-dome has now been named LED_dome, and the Mode Selector box mode_selector_box. I also made a rule for the Zed stereo camera as this will hopefully be implemented in the not too distant future. The udev rule file has been saved in /etc/udev/rules.d/ as 90_cyborg_usb_rules.rules . If any of the devices is switched out, the rule for that specific device needs to be altered to be able to find it.

6.2 Creation and Update of Wiki on GitHub

The wiki page on GitHub has not been used for previous work on the cyborg. There are many reasons for this. People working in repositories from their own GitHub page, every node has had its own repository, and possibly because people generally do not think writing documentation is fun. This year though, we have tried using the Projects page on GitHub to organise, instead of using Trello. Using this means that we can connect issues to different projects as things to do, and is generally very similar to Trello. Also, all currently active repositories have been added to one big repository which contains all code for the cyborg. Having one instead of several makes it easier to know which is actually in use. Each student then makes a branch to work on and merges that with the master when what they are working on is implemented and tested.

As the wiki on GitHub is still undergoing updates, it is my suggestion that the wiki contains references to where each author wrote about their implementations as well as detailed information about the code and how to change or update it. This means that there will be overlaps and the same information both places, but it is my view that the wiki should contain the information and documentation needed to start using the code, and if more reading is needed, then it can be found in the referenced theses and reports. The other wiki [30] at NTNU's servers should then contain overall information about what the project does, though not as thorough as [23], and information about the hardware and software used on the cyborg. This way there will be a logical separation between the websites and what can be found there. In addition, the project uses Box as a closed drive to store previous project reports and masters theses, among other things such as images, videos and sample MEA data to mention some.

6.3 Making a power cable for testing of the LED Dome

Earlier there existed a power cable used for testing, but over the summer that has either been misplaced or thrown away. As further development of the LED dome was planned and we were several students working on the cyborg robot, there was a need for another power cable. Making this cable became a nice introduction to some of the hardware on the cyborg. Especially how the LED-dome is structured. Finding the right papers and documentation on the robot to read to find more information.

The main reason the cable is needed is that the ESP-32 cannot provide enough power from its pins to power the 791 LEDs on the LED-dome. The LED strips require a 5V supply and can draw several amperes. Meaning that the cable has to be sturdy enough to be able to handle up to around 10A. Though this is not a cable for continuous use in

Figure 6.1: Power cable used in testing the LED-dome away from the robot.

a home, and 10A is in the rarer ranges of uses. The robot should be able to provide 30A which is more than enough. During testing the LED-dome has drawn up to around 10A.

The connectors on the LED-dome are of the type 2-pin Molex, the female being on the robot and the male on the cable. Meaning that the new power cable needed the female connectors. They were hard to find, and as I did only have male connectors and did not want to wait another week on ordering and shipping, a little ingenuity was needed. At Omega Verksted they had female 4-pin Molex connectors with a 90-degree angle on the conductive part sticking out at the back. As only two of the pins were needed I removed the excess pin's conductive parts, and straightened out and cut the two others to the right length. This meant that the male connectors could be fitted on both sides of the female connectors. Fortunately, the pin terminals on the male connectors were just the right size so that they gave a tight fit, and does not need to be secured with tape or otherwise. As this is to be used for testing on a desk, I did not see the need to try and secure the cable more than necessary. The new cable can be seen in fig. 6.1. The black coloured cables are naturally for ground, while the orange is power.

6.4 Refrosting the LED dome

The LED-dome is made from a clear plastic material called VIVAK PETG, but as it was unwanted to be able to see the insides, the dome was frosted by an EiT group in the spring

of 2019 [1]. They tested several different methods. First, they used sandpaper, but the method was discarded for being too coarse. Another method was sandblasting, while it gave a smoother finish it also looked dirty and not intentional and was also discarded. The last option was a frosting spray which gave satisfactory results but was easily scratched. As such, they tried applying a layer of clear lacquer on top. Unfortunately, this resulted in the frosting spray turning clear again. Ultimately they ended on wet sandpaper with grit 150. Their results can be seen in the report mentioned above. With this in mind, I took a turn on refrosting the LED dome. All of these tests, mine and by the EiT group were tested on scraps and then the chosen method was used on the inside of the outer dome. My first take was to try sandblasting again, as it might look different on a bigger scale than 10x10 cm area. The next idea was googling other methods of frosting something. The Google search came up with a lot of different frosting sprays for use on windows. But seeing as they are meant to inhibit looking in from the outside, they are quite thin and things close to the window would be possible to see from the other side. Which is the main problem of the LED-dome. The LED strips are closer than a centimetre from the outer dome. This thought process gave me the idea of using a less transparent spray, like the white used for painting the body of the cyborg.

Figure 6.2: Testing of white spray-paint on a piece of Plexiglas. The right side has a coat of paint thicker than the left.

I used two pieces of Plexiglas to test different methods of spraying with white paint. The piece in fig. 6.2 was first sanded with 240 grit, wet and dry, and then painted. The right side can be seen to be a shade whiter than the left, the reason being a thicker layer

Figure 6.3: Testing of white spray-paint on a piece of Plexiglas. The left side was painted and then dabbed with paper, and the right side is paint with a coat of transparent spray on top.

of paint. Testing with an already sanded piece gave an idea of what the dome might look like when painted as it is already frosted with sandpaper. The other piece, seen in g. 6.3 was painted the same way on both sides and then dabbed with some paper on the left, while the right got a coat of clear lacquer on top. As seen from the image the dabbing did not work very well, and the extra coat of lacquer might have helped protect against scratches but did little else. It was therefore decided that just painting would be satisfactory. If any bubbles or imperfections should occur during painting, as long as they are not signi cant, they would only be possible to detect from the inside or when holding the dome up to a bright light source. This because the painting was done on the inside.

After giving the dome a coat of white and holding it up to the light it is possible to see that the paint is not entirely even. This should not be a problem in use and is impossible to spot when the dome is assembled. Without buying another can of paint to add another coat, there is not much to do about the unevenness. In the after image, we cannot see the LED strips, only each individual light. If one wished to not be able to make out each individual LED I can see two options. Either lay additional layers of white paint, but risk it being too little opaque and needing to turn up the intensity of the LEDs draining the battery faster. Or increasing the distance between the two domes. The light will then blend better making it less sharp. An issue with this is that it is not possible to increase the distances everywhere without making a new dome. That is just