UNIVERSITY OF APPLIED SCIENCES RAPPERSWIL

Environment Mapping Self-Sustainable Robot

Semester Project

David Grob and Daniel Krüsi

Advised by Prof. Dr. Joachim Wirth

December, 2008

Abstract

The aim of the emss *project is to tackle the problems of a self-sustainable, environment mapping robot in a bottom up approach. The major challenges imposed by such a mobile robot include the assembly of hardware, as well as the necessary software algorithms for localization, navigation, and discovery. Using ready-made hardware, much of the time-consuming electrical engineering problems have been avoided, allowing a strong focus on software. In addition to generic theoretical solutions to problems such as smooth curve based navigation, movement emulation, and finite space discovery, we present a software framework for the control of a two-wheeled robot. The* emss *framework, consisting of a set of "hot-swappable" components, provides an extensible design, which allows a wide array of functionality and supports different strategies for the same problem. Furthermore, the softwares library offers a set of graphical user interface widgets, which can be easily combined to create a rich interface with the framework.*

Table of Contents

1 Management Summary

1.1 Current Situation

The field of robotics is a well studied area in engineering, mathematics, and even distant practices such as philosophy. Mobile robots, those which are movable and disjoint from their environment, have been keenly studied by both the research community and hobbyists. There remain few untouched topics revolving around the problematics of such robots. However, we feel there is significant room for creative ideas and new insights in the areas of self-sustainability and environment mapping. Because each robot potentially sees the environment different from its peers (different sensors compose different perceptions), the possible angles of approach vary strongly between each project.

The *iRobot Create* is a special version of the popular vacuum cleaner robot called "Roomba". The *Create* is specifically designed for hobbyists and robot enthusiasts who wish to build on an existing platform instead of starting from scratch. With the release of the *iRobot Create* model, an eruption of robotics projects sprung up, most which were based on the *iRobot Create Command Controller* – an elegant (but underpowered) microcontroller. However, a small niche of *iRobot Create* projects had a laptop-centric approach (Jonathan, Jonathan, Arvind, Omair, & Sukhatme, 2008), where basically a full-blown computer is strapped to wheels. Such an elegant, yet powerful, design proves very attractive, as it is low-cost and yields results quicker than a classical construction. Our implementation of the *emss* robot using the Create platform follows this design, seeking low-cost fast results. Because of the young age of the *Create* product, there is little code available to kick start interested enthusiasts, hobbyists, or researchers with their project. The same applies for design concepts, where the literature revolving *iRobot Create* projects widely vary in their opinions of design advantages and disadvantages (Kostandov, et al., Robot Gaming and Learning using Augmented Reality, 2007) (Martinez, Haverinen, & Roning, 2008) and does not provide clearly extractable strategies.

1.2 Requirements

The goal of the *emss* project is to tackle the problems of a self-sustainable, environment mapping robot in a bottom up approach. The requirements of this approach include the realization of the following tasks:

R 1	Assembly of hardware
R 2	Communication between hardware and software
R 3	Robot control by human interaction
R 4	Robot control by navigational structures

Table 1-1: Requirements

An additional requirement which covers the entire project is the research and study of solutions to various problems presented by the different tasks. This requirement calls for the attention to theoretical problems in mathematical fields with application in robotics.

1.3 Approach

Throughout the project, a bottom up approach was applied. Fundamental tasks were completed first, each task building on the foundation of the last. At the project start a considerable amount of energy was invested in preparing the basic hardware components. The *iRobot Create Platform* was significantly modified to support the needs of an x86 laptop, which included the modification of the charging station, as well as the framework required to support the weight of the laptop. A serial controller for the C++ *Qt* Framework was written. The later project phases included the implementation and analysis of a testing interface, which later evolved into the *emss Interface*. Test results were gathered and analyzed, helping to form the basic ideas for the map-building and navigational algorithms. A controller component was developed to send steering information to the robot and receive sensor data from the robot. The next phases focused on the implementation of the robot's core functionality, including navigational skills and a map building algorithm. With the aid of these base components further tasks were created. For this project, navigation tasks and discovery tasks were realized. A considerable focus was placed on theoretical solutions and their application in software. This included the understanding and application of existing mathematical concepts, such as spline curves, as well as the development of private concepts.

1.4 Results

1.4.1 Hardware

The realization of the *emss* hardware played an important role in the *emss* project and was successfully executed in the initial phases. The original proposal (Krusi & Grob, 2008) called for a two-wheeled robot with a mounted x86 laptop, and was achieved by assembling an *Averatec 2100* laptop onto an *iRobot Create* kit. The mounted laptop was attached to the *Create* using a custom-made lightweight frame, raising it slightly above the chassis of the robot body. Additional power lines were added in order to allow the laptop to recharge itself when the *iRobot* is on its charging station. External sensors, such as an IR Range Finder and VGA camera, were also assembled and connected to the appropriate interfaces.

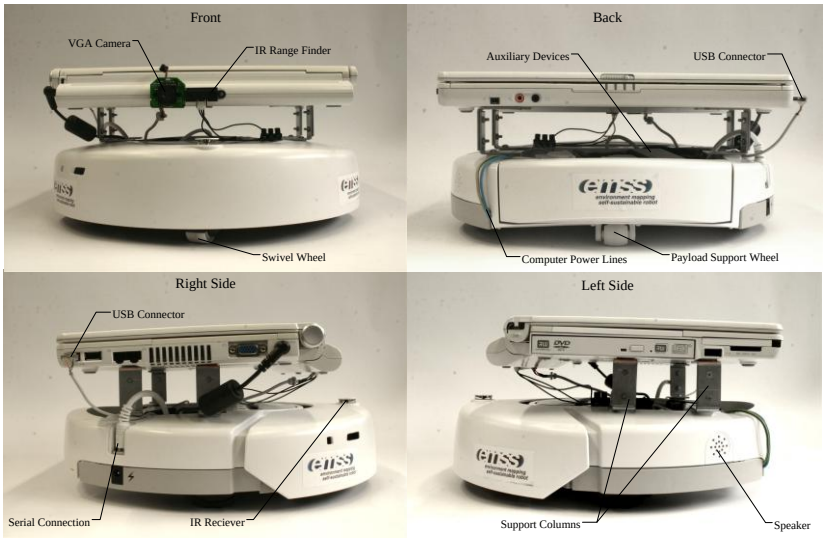


Figure 1-1: Assembled *emss* Body and Chassis Hardware

1.4.2 *emss* Core

The *emss* Core constitutes all the fundamental components for our *emss* robot. The modular framework allows the implementation of differently flavored components, as well as providing an extensible environment. The Core supports the separation of execution, where needed, realized by process threads. The implemented components include multiple Controllers, environment Maps, a Movement Tracker, Tasks and a Task Manager, and a Navigation component. Noteworthy accomplishments include the communication with the hardware, robot control by joystick, and the navigation through way points using smooth curves.

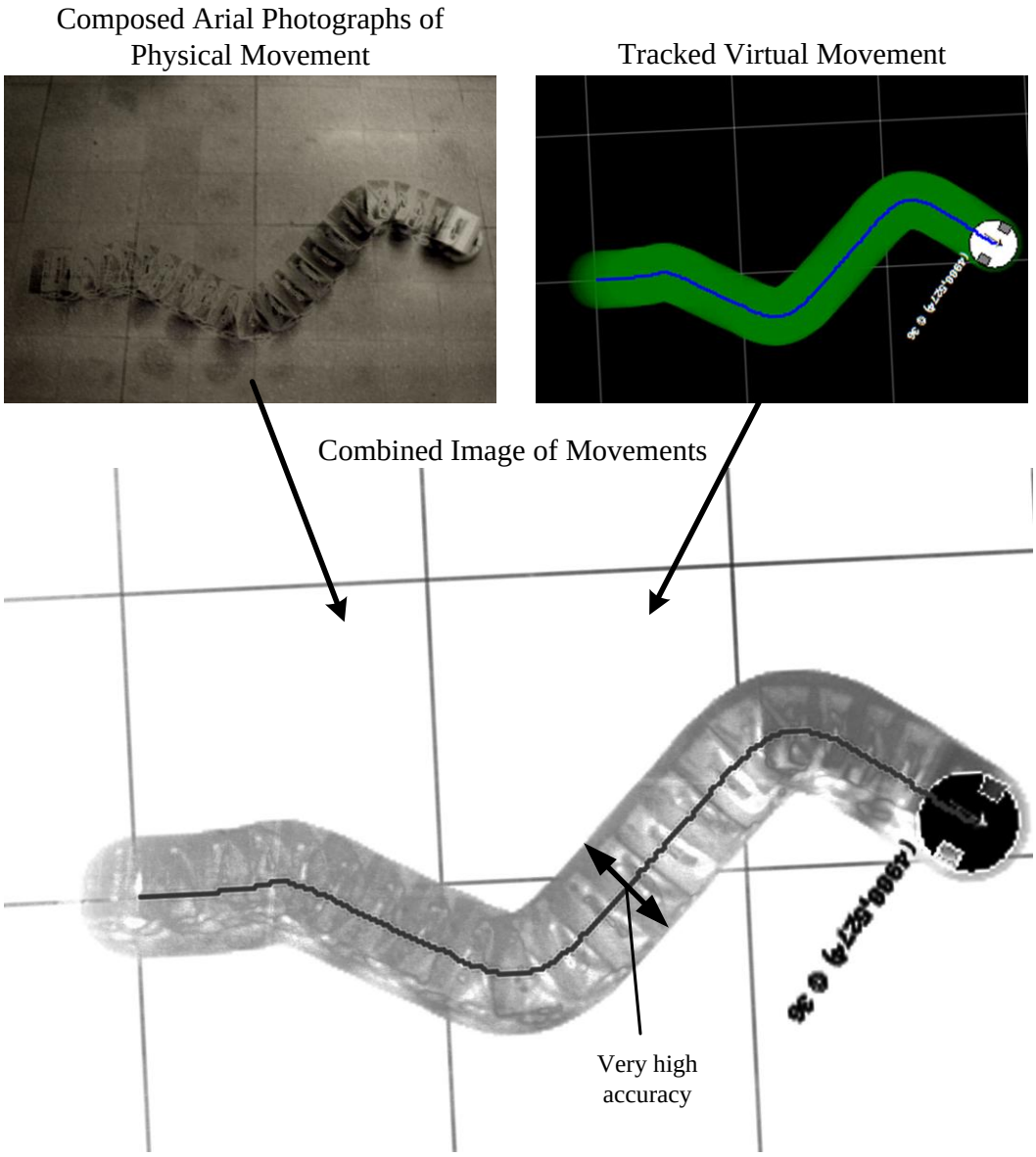


Figure 1-2: Successful Navigation of a Smooth Curve

1.4.3 *emss* Interface

The Interface application *emssInterface* was developed as graphical user interface to control the *emss* robot and other Core components. This application is divided into three groups: *Connection*, *Controller*, and *Data*. The Connection group allows one to connect and disconnect with the *emss*

hardware. It supports the selection of all the implemented Core Controllers and Movement Trackers. The *Controller* panel features various tabs each designed specifically for the individual Controllers. The important diagnostic panel *History* is located inside the *Data* group box. The history widget provides all necessary diagnostic and debugging information which streams from the various Core components. Along with the *History* panel, other data sources are provided such as *Serial Port* and *Sensors*. The *emss* Interface allows multiple instances of Core Viewports, as well as modal windows such as the Task Editor and Settings Editor of both the Core configuration settings and the Interface settings.

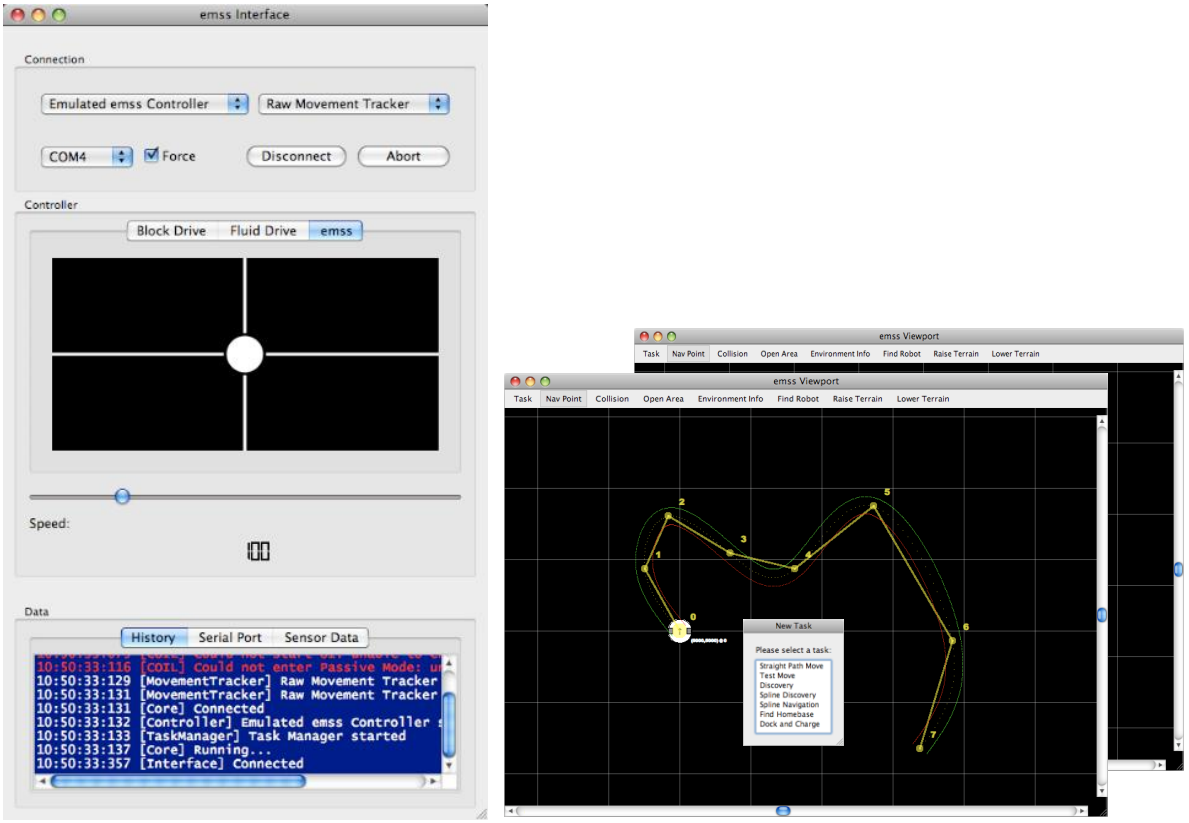


Figure 1-3: *emss* Interface with Multiple Viewports

1.5 Future Outlook

Because of the projects extensible nature, numerous additions and expansions can be imagined. Furthermore, the current Core components can be used to perform more concrete Tasks, such fetching objects by remote control. The x86 nature of the processing power also leaves room for a wide array of additions, including the addition of external sensors. This opens the *emss* project to many different future possibilities, spanning from visual computing to human-aid robotics to research environments.

In addition, there is still much room for improvement and optimization to the current algorithms and components. The continuation of the *emss* project is important in order to maximize the profit / investment ratio, as many of the resulting benefits and applications are yet to come. The Bachelor project will enable these improvements, as well as provide the resources for further application of the *emss* robot.

2 Introduction

Robotics (rō-bŏt'ĭks). To most people this singular noun with Russian roots[1] causes a spark in the imagination leading to some sort of alien emotion the human has not yet gotten completely familiar with. Whether the emotion points towards interest, fascination, hunger for understanding, curiosity, or even fear, it most certainly seems something quite new; something of the 20th century. *Seems* however cannot be stressed enough, as robotics is nothing new, yet alone something of the 20th century. Robotics goes back well beyond hundreds of years, where Arab inventors, such as the Mesopotamian scholar Al-Jazira, were hard at work. The creation of a programmable humanoid band of robotic musicians by Al-Jazira (Fowler, 1967) leaves a startling impression on the history of robotics. Followed by numerous inventions, the Jacques de Vaucanson's *Digesting Duck* (Wood, 2002) shows us the well-developed hunger humans have to breathe life into their creations. This brings us to the mainstream understanding of robot, a term not even a hundred years old (Čapek, 1921). However, in our time robotics has rooted itself with a new purpose, drifting from the fantastical humanoids and casting anchor in the industrial arena as a faithful servant to mankind.

Figure 2-1: Vaucanson's *Digesting Duck* (left), Snapshot from *Leave It To Roll-Oh* (right)

The modern robot, navigating far away planets and welding circuits with micro-meter precision, is far from the 1940's romantic idea of robotics depicted in *Leave It To Roll-Oh* (Handy, 1940). Robots don't serve us breakfast in the morning; neither do they take our hat or coats from us when we return home in the evening. Furthermore, they don't seem to clean our house while we are gone. Here again, the word *seem*, should be examined closer. We see that while we are gone, robots clean our dishes, wash our clothes, and more commonly even vacuum our floors and cut our grass. Maybe the progression of the robot has taken a more subtle path than originally imagined. Nonetheless one could argue these are nothing but machines performing trivial pre-programmed tasks – true depending on your perception of what constitutes a robot. We will let the reader decide.

So, why robotics? What is so significant and interesting about automated machines? Naturally, there are many different answers coming from many perspectives, but in our opinion the answer has something to do with mankind's (yet to be?) greatest birth: the artificial being. The significance and importance of creating life needs not to be discussed further (we assume), so in turn we feel creating

[1] The Slavish word *robota* is the root of the modernly coined term *robot*. Its Slavish meaning can be translated to *work* or *forced labor*.

an artificial being of intelligence holds the same, or at least similar, importance. However, a robot in its essence does not constitute an artificial being at all, as it is just a pre-programmed machine, but we recognize that it is a subset of the embodiment of artificial intelligence, and surely plays an important role. Furthermore, as mankind continues to strengthen their reliance on machines, the field of robotics will become increasingly important. Understanding how machines work has been finely crafted over the centuries through the aid of mechanical and electrical engineering, et al. Now more than ever it is time to explore the behavior of machines. What better way to do so than in the field of robotics? Is this not the study, development, and application of machine behavior? This, however, is not a proposal for building a Robotics and Artificial Intelligence lab, but rather a teaser and justification for our passion in robotics, so let us continue.

A fundamental problem in robotics is the perception of the environment. To us humans, this is a trivial task. We see, hear, and touch our environment around us, combining these senses to form a somewhat indescribable organic perception of the world (Thorpe, Clatz, Duggins, Gowdy, MacLachlan, & Ryan, 2001). In order for a robot to successfully, and flexibly, function in a shared environment, it must be able to perceive it first. For thousands of years we have tried to understand and imitate our perceptions, starting with primitive cave drawings and slowly mastering perception with realistic paintings. More recently, man has continued this practice by creating photographs, audio recorders, and motion pictures. However, mimicking perception is a different dimension than understanding perception. A robot can easily imitate its environment through the form of radar, sonar, lidar, or visual images, but understanding it has proven (surprisingly) difficult. None-the-less, the successful perception of the environment is an important prerequisite to many functions of a mobile robot. Being able to stop before driving down stairs, avoid hitting walls, and plan efficient paths through space all involve the mapping of the physical environment through the robots perception.

One of the greatest restrictions robots face today is lack of self-sustainability. Most robots rely on direct external influences to remain “alive”, such as the exchange of battery packs or connection to a power supply. A robot that advances progression must prove to be self-sustainable. In other words, this means that a robot must be able to take care of itself, providing the necessary resources it needs, without external help. Looking at the classical design of a mobile robot, consisting of wheels, motors, and processing units, the underlying resource common to all components is electricity. In mobile robots this resources is stored in a battery, which the robot consequently must carry with at all time. When the battery is depleted, the game of life is over and the robot dies. To live, the robot is reliable on an external influence, which consequently cares for the robot and recharges it. Becoming self-sustainable requires such a robot to care for itself, recharging when necessary. This task is not so trivial, and in our human environment moderately complex. Such a robot must master the art of navigation, finding its way back to a power source when needed. But the rewards are large; it is the first step to self-sustainability for the modern robot.

3 Hardware

A robot's hardware is the physical embodiment of its being (i.e. software) in the environment. With hardware, a robot perceives the world and ultimately causes change of state therein. More importantly, the hardware typically enables, but also limits, the robots capabilities to a certain set of functions, tasks, and observations. Thus, when designing and building robots, one must understand the consequences (both beneficial and disadvantageous) of the selection of hardware. Furthermore, when creating a mobile self-sustainable robot, one must consider the power consumption for each connected component. Finally in a more general scope, every additional component typically calls for additional processing resources – an important factor to eye when building a robot.

The *emss* hardware is a simple set of ready-made components, deliberately chosen to avoid many of the tedious electrical engineering problems presented in the field of robotics. All the components are plug-and-play and off-the-shelf available as consumer electronics. The hardware consists of two groups of components: the body and chassis, and the charging station.

3.1 Body and Chassis

The mobile hardware of *emss* consists of the ready-made *iRobot Create* platform and a 12-inch x86 laptop computer. In addition, a proposed auxiliary tower has been developed for containing additional sensors, however this has not yet been implemented, as seen in Figure 3-1. For a complete listing of hardware, please see section 7.2 in the Appendix.

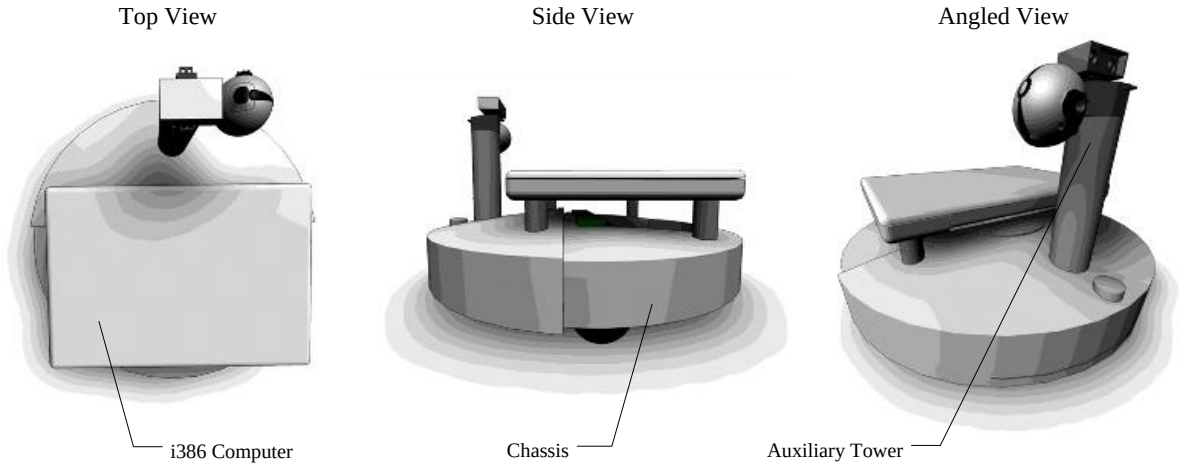


Figure 3-1: Body and Chassis Concept Renderings

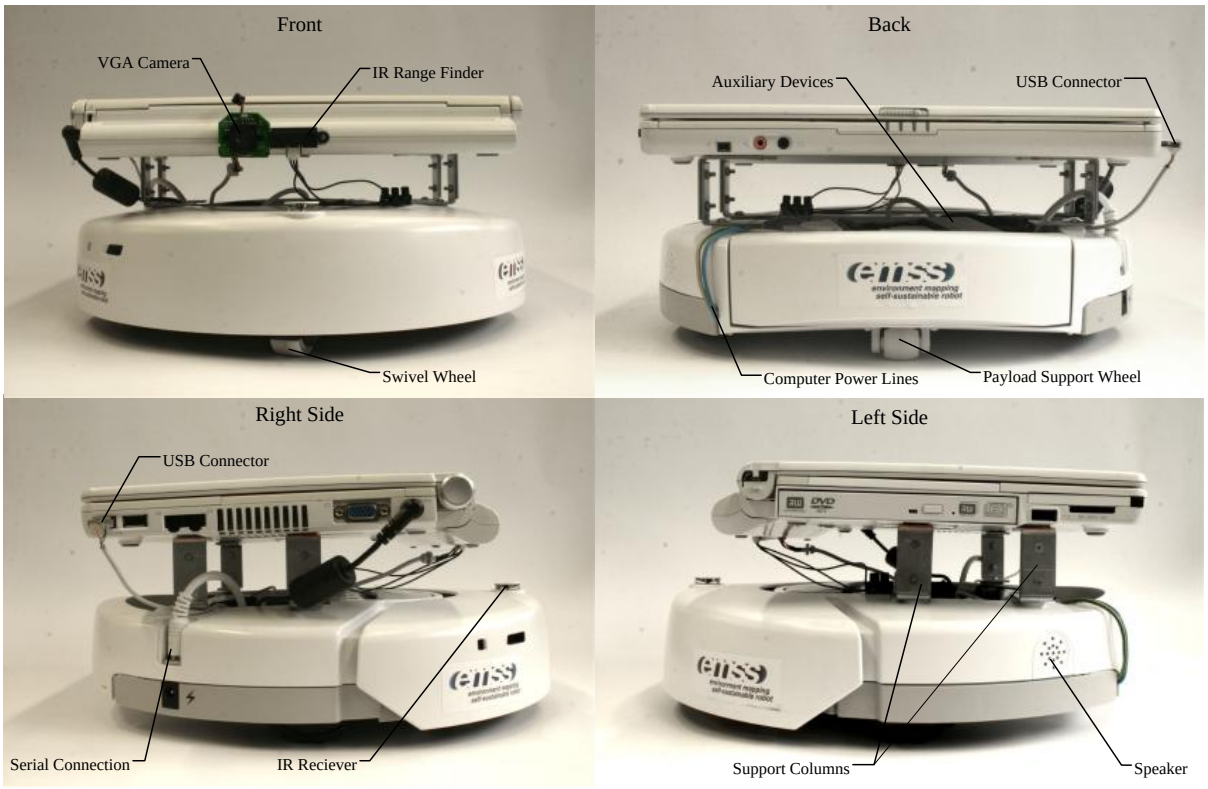


Figure 3-2: Assembled Body and Chassis

3.1.1 iRobot Create

The circular *iRobot Create* platform has two wheels with precision servos offset at the radius of the chassis and a swivel wheel located at the front. The robot's center of gravity is slightly offset towards the front by the battery payload, allowing the chassis to balance on three wheels. Included internally within the *iRobot* is a small programmable microcontroller which directly accesses the different sensors and controls the servos. In addition, the controller monitors the robot's hardware health and can prevent damage by preemptively stopping movement of the servos.

The two-wheel design creates a differential steering system, a common design concept in many mobile robots. Such a system allows the greatest flexibility with the least moving parts. Various rotations, even around the center axis, can easily be executed by independently driving each wheel at different speeds, which enables a differential steering system's largest advantage of not needing a separate steering device.

The built-sensors span the categories of touch, visual, and self-awareness. The touch sensors include three digital contact sensors located directly behind the bumper. The bumper is designed as to allow each sensor to be individually triggered upon collision. This behavior proves useful when resolving straight-on collisions with obstacles as the individual bumper sensors may be used to determine the course of action. All visual sensors are in the form of infra-red light emitting diodes and receivers. The receiver located on the bumper is used for precise alignment with the charging station. This receiver can detect different infra-red wavelengths. Emitters and receivers located inside the bumper behave as "cliff sensors", disabling the robot from driving down stairs or other sudden drops. The servos provide analog feedback to the controller in the form of rotational axle position (iRobot

Corporation, 2006), providing the robot with an indirect form of self-awareness about its current state. The information provided by the servos to the controller and finally to our software is vital for localization.

Some modifications to the chassis were undertaken to accommodate our needs. The most important modifications are the computer contact points for the charging station along with the necessary cabling (shown in Figure 3-3). These are used in addition to the built-in contact points for charging of the *iRobot Create* batteries along with the computer batteries. Both sets of contact points expose 12V power supply lines which can connect by physical contact to their peers located at the base station (see Figure 3-6). This mechanism allows the *emss* robot to easily drive and mount itself on top of the charging station, creating connections between all sets of contact points, and ultimately recharging its batteries.

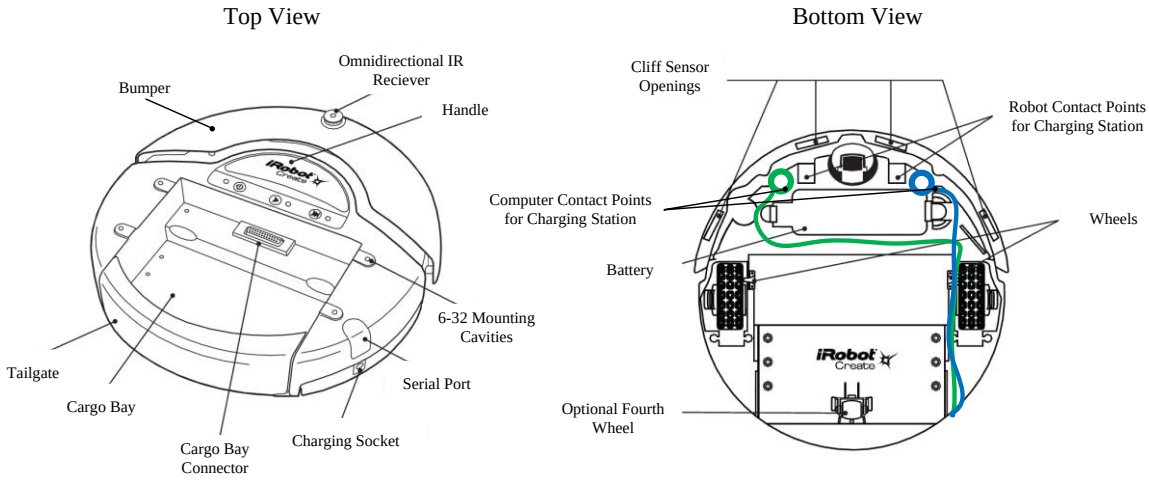


Figure 3-3: iRobot Create Platform Schematics with Modifications

The decision to use the *iRobot Create* platform as the basis for our robot was natural. The platform not only saves a significant amount of time which would otherwise have been spent solving electrical engineering problems, but also provides a robust and well tested hardware basis which has also been proven to work well in many similar projects (Yen-Chun, Yen-Ting, Szu-Yin, & Jen-Hua, 2008) (Nathan, 2007) (Kostandov, Schwertfeger, & Jenkins, Toward Real-time Human Detection and Tracking in Diverse; Mataric, Koenig, & Feil-Seifer, 2007) (Takacs & Hanak, 2008) (Martınez, Haverinen, & Roning, 2008) (Kelly, Binney, Pereira, Khan, & Sukhatme, 2008). Furthermore, the size and shape of the platform suits our need to mount a computer and other peripheries very well. Finally, the affordable total cost of ownership of the *iRobot Create* platform allows us to build a full blown robot with a small budget.

An additional, more powerful, microcontroller was purchased, but we decided not to use it as we could control the *iRobot* sufficiently using a serial interface connected to the computer. This however has proven to come with some difficulties, as well needed additional sensors cannot be directly connected to the *Create* platform[2].

[2] The *iRobot Create* provides only one single analog input, which is used by the infrared range finder described in section 3.4.5.

3.1.2 x86 Computer

The on-board microcontroller mentioned in section 3.1.1 is only used to relay commands as its capabilities and resources are extremely limited[3]. Thus a full-blown x86-architecture laptop is mounted onto the *iRobot Create* to provide the brain power for *emss*. Having an x86 computer provides many advantages over a microcontroller, especially at later stages in the *emss* project when more complex sensory devices such as video cameras are added. This is consistent with our vision – to provide a robust platform for future extensions.

The computer is mounted directly on top of the *iRobot* chassis by four metal legs with 3 cm of vertical space in-between. This space is needed in order to allow a scanning degree greater than 180 degrees for the infrared receiver mounted on the chassis bumper. Because the computer is used as a microcontroller for the *iRobot Create* hardware, there is no need for a screen and the laptop screen is kept closed during operation. Just a mere 30 cm wide and about 2 cm thick, the computer weighs about 2.5 kilograms (Chan), a payload which can easily be handled by the *iRobot Create*.

The processor runs at 800 megahertz[4] with one gigabyte of physical memory available. The laptop computer has three USB 2.0 ports, one of which has a serial port adapter attached to it. The serial interface of the *iRobot*, described in section 3.3, is connected to the serial port adapter, which in turn is connected to the USB bus of the computer. A custom USB adapter was made in order to cut down on the size and extrusion depth on the side of the laptop, shown in Figure 3-4.

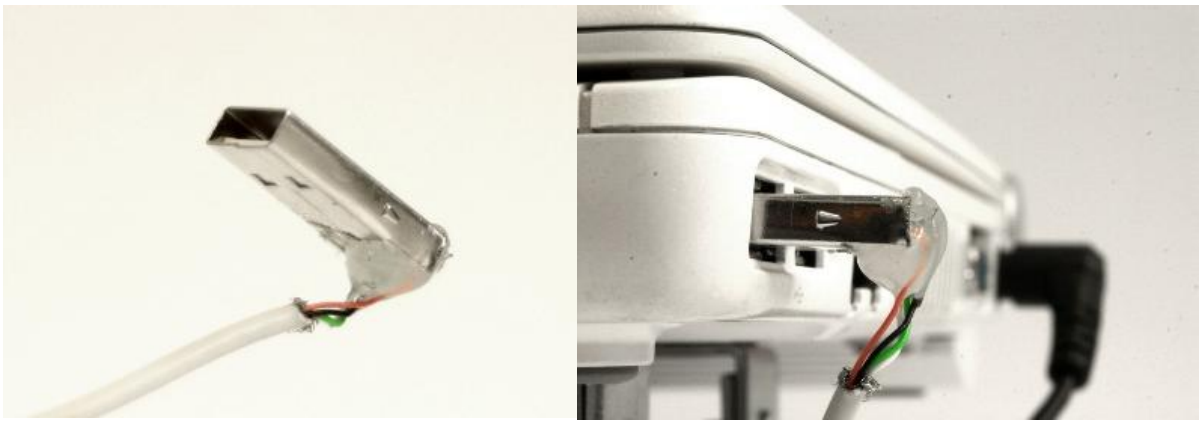


Figure 3-4: Custom USB Adapter for Auxiliary Devices

3.1.3 Auxiliary Tower

The auxiliary tower serves as a platform for adding additional sensory equipment and other accessories. Typically these devices would be connected to either the *iRobot Create* or the x86 computer. Although not realized as in the original concept, the auxiliary tower has found itself spread out across the chassis and body with the intention of being implemented at a later time. Current accessories include an infra-red range finder, a Wi-Fi USB device, and a light-weight VGA USB camera.

---

[3] Programmable memory in the on-board controller is limited to 100 bytes (iRobot Corporation, 2006).

[4] The processors natural speed is 1.6 GHz but has been clocked down to 800MHz in order to reduce battery consumption.

During the implementation process it was quickly understood that the *iRobot* built-in sensors would not be sufficient for rudimentary environment mapping. Thus an analog Sharp Infra-Red Range Finder (*Sharp GP2D12*) was added. We chose this device because of its excellent reputation in robotics as a trustworthy and affordable range finder. This sensory device uses the one and only analog input available on the *iRobot Create* to output its data. The IR Range Finder can detect objects up to 80 cm away and remains accurate until 10 cm close (Sharp, 2002). Currently, the IR Range Finder is located in the front of the robot mounted on the forward side of the laptop computer (Figure 3-2, Figure 3-5). It is elevated at about 14 cm and points directly in the forward direction of the robot.

Because of repeated problems with the computer's internal Wi-Fi card, an external Wi-Fi USB device was added to the robot for better Wi-Fi reception and performance. The problems with the internal device arose from the fact that the laptop screen is closed during robot operation. Most laptops house their Wi-Fi cards directly underneath the keyboard, and when the screen is closed it acts as a strong shield against any incoming signals. The external device has much better performance as it is not shielded by various enclosures, and, if needed, may also be freely placed on top or on the side of the chassis. Currently, the external Wi-Fi device is located in the cargo bay as shown in Figure 3-2 under "Auxiliary Devices".

Although the envisioned *emss* software does not call for a camera device, a very small light-weight 365gm VGA Camera (Logitech, 2006) has been included for external purposes. The *Logitech QuickCam* acts as a standard PC Video Device, compatible with most operating systems, and could easily be integrated into the software at a later time. Currently the camera serves only as a reference device when remote-controlling the robot.

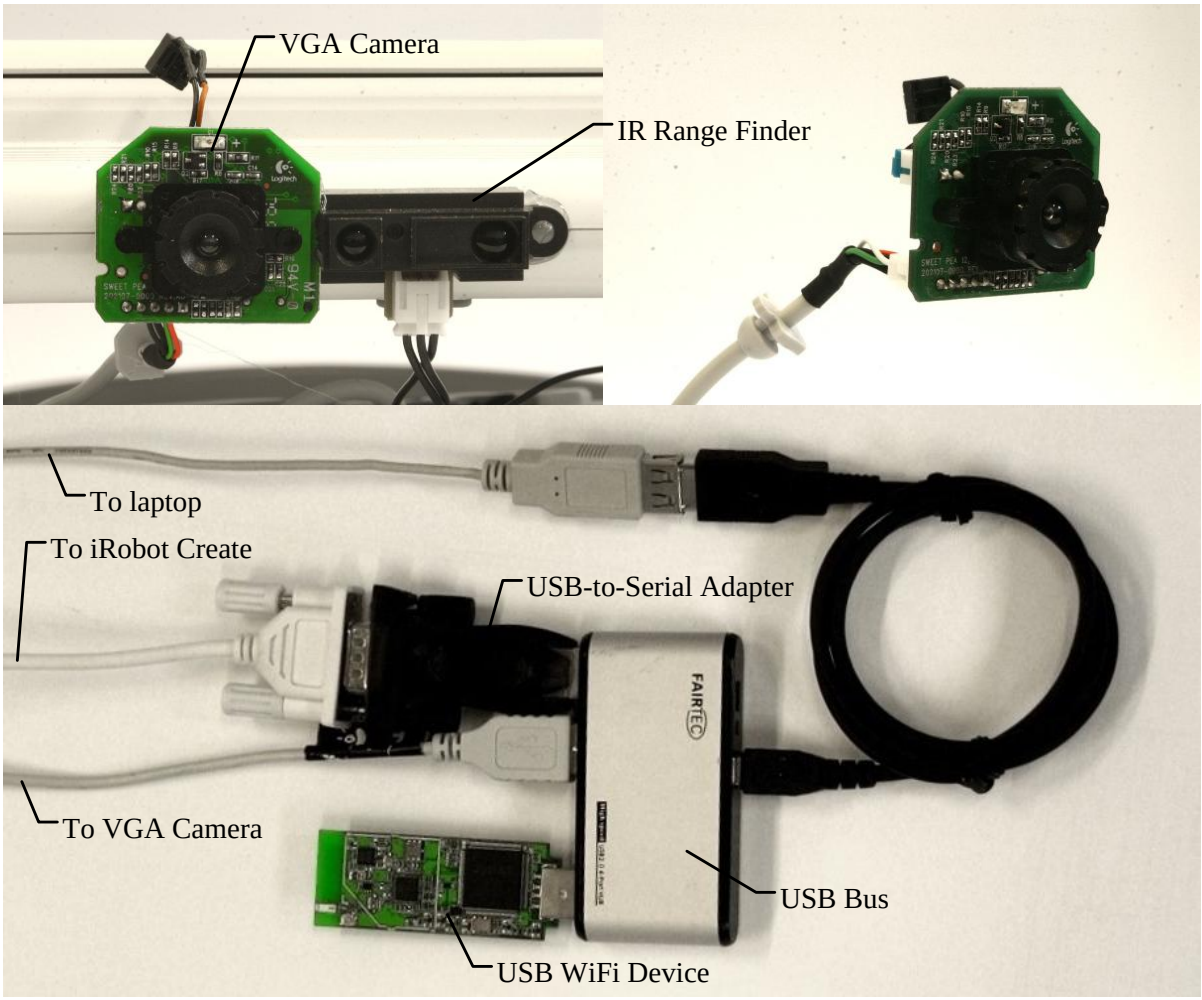


Figure 3-5: Auxiliary Devices and Connection Layout

3.2 Charging Station

To fulfill our vision of creating a self-sustainable robot, *emss* must be able to recharge itself without any external help. The supporting hardware for this must therefore be robust and encourage simplicity in order to minimize possible faults. There are two sets of batteries which require recharging: the *iRobot Create* batteries and the laptop computer batteries. *iRobot* provides a great docking and charging system for most of their products, which we used as the basis for our charging station.

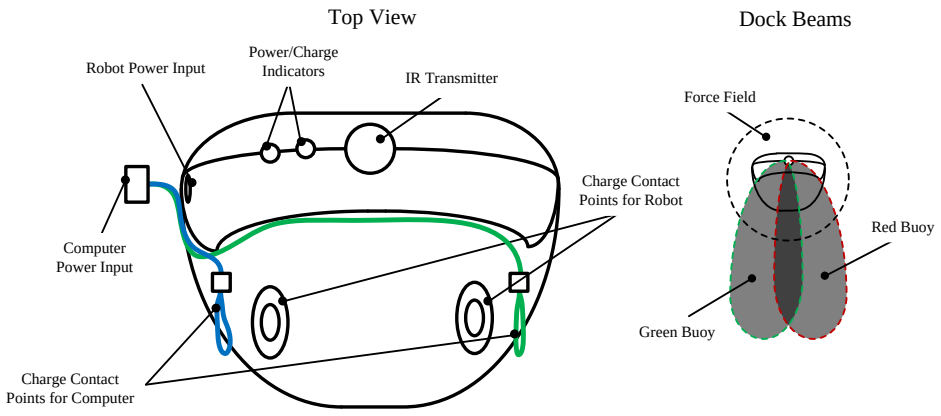


Figure 3-6: Charging Station Schematics with Modifications

The *iRobot* system requires the *Create* robot to find the station using the top-mounted infra-red receiver. The infra-red emitter located on the top of the charging station (see Figure 3-5) sends two separate beams at different wave-lengths, allowing the robot to align itself flawlessly with the charging station and drive directly onto it. The robot alignment is very simple as the robot can determine easily based on which dock beams it sees whether to peer left or right until it is centered where it can see both dock beams. When the robot successfully drives onto the charging station, the contacts described in Section 3.1.1 and shown in Figure 3-3 make physical contact with the contacts shown in Figure 3-5.

Our current construction presents some problems. The *iRobot* charging station is quite a steep climb to dock[5], which causes trouble for our robot at full payload. This can be corrected by re-constructing the charging station chassis using the existing internal components. Additionally, the added contact points for the computer power lines are not very sturdy, and after several docks they tend to loosen and the contact quality is degraded. This also can be corrected by mimicking the *iRobot* contact points with the appropriate hardware.

Figure 3-7: Assembled Charging Station

[5] About 10%.

3.3 Interface

The *emss* interface is split into two groups: User Hardware and *emss* Hardware. User Hardware consists of the user and remote terminal for controlling the robot and displaying diagnostic information[6] while *emss* Hardware consists of everything attached to the mobile robot. These two groups must also naturally be able to interface with each other. As our robot controller boasts x86 hardware with a WiFi device, we chose the obvious connecting protocol of TCP / IP between User Hardware and *emss* Hardware. The final important interface is between our x86 computer and the *iRobot Create* hardware. This is realized over a serial connection using the *iRobot Create Open Interface*[7] protocols. It should be noted that the interface between the *iRobot Create* and the actual hardware is of no interest to us, as it does not expose any significant features we cannot access over the *Open Interface*.

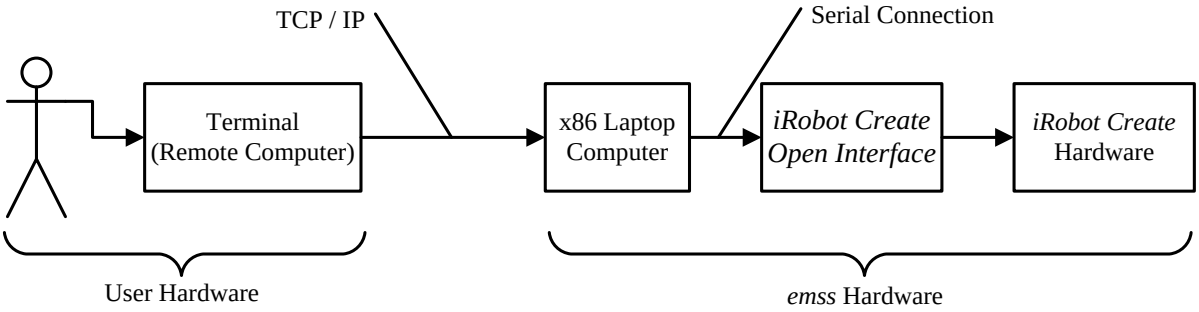


Figure 3-8: Interface Pipeline: User to Hardware

The *iRobot Create Open Interface* consists of an electronic interface and a software interface for controlling the robots behavior and reading its sensors. The electronic interface includes a 7 pin Mini-DIN connector and a DB-25 connector in the Cargo Bay for connecting hardware and electronics for sensors and actuators such as a robotic arm or light sensor. We chose the 7 pin Mini-DIN connector for communicating with the *iRobot*. The software interface lets you manipulate *iRobot*'s behavior and read its sensors through a series of serial commands over a standard serial interface[8]. The *iRobot Create Open Interface* is well documented and has been extensively tested by the *iRobot Create* community. While there are some features missing, the interface proves to work well for our needs.

3.4 Performance

At the beginning of the *emss* project the performance of the hardware was largely unknown. Certain metrics such as maximum speed, et cetera, were given but parameters such as accuracy of sensors are not officially documented if at all mentioned in external literature. As soon as we had a working controller to send commands to the *Create* serial interface we performed some tests to build an idea of the performance and accuracy of *emss*.

---

[6] Currently this is the Graphical User Interface frontend to our software.
[7] Commonly denoted as COIL.
[8] TTL levels of 0 to 5V.

3.4.1 Processing Speed

The x86 laptop processor is clocked to 800 Mhz to conserve power. This speed allows us to fully run a host operating system with our software on top. Several time-critical processes with millisecond execution intervals can easily be run within our framework, with additional GUI processes running alongside. Important processes, such as the controller process who sends and receives information on the serial interface, are not run in real time, neither are they run in soft-real time[9]. There is no guarantee of the process run loop interval. The commands sent over the serial interface are sent at baud 57600.

3.4.2 Battery

There are two battery components in the *emss* hardware: x86 laptop power supply and *iRobot Create* power supply. The laptop battery has a run time of circa 3 hours. The *iRobot Create* can run for 3+ hours at normal[10] speeds. Therefore the *emss* runtime is restricted by the laptop battery at 3 hours.

3.4.3 Movement Speed

Each wheel can independently move at ground-relative speeds ranging from 20 mm/s to 400 mm/s. Thus, the maximum linear motion of *emss* is capped at ±400 mm/s. Furthermore, emss can turn around its center axis at ±400 mm/s along the diameter of the wheel offset. We found that speeds less than 50 mm/s proved inaccurate.

3.4.4 Movement Accuracy

Movement accuracy is important for the localization of the robot. If the accuracy was at 100%, an emulation of the robot movements would be sufficient for localization as long as no external influences were acted on the robot. Controlling the robots movement is achieved by (independently) setting the left and right wheel speeds. The *iRobot Create* platform provides servo feedback measurements in the form of Δs and $\Delta \alpha$ where Δs is distance travelled in mm since the last measurement, and $\Delta \alpha$ is the change of angle of the robots orientation since the last measurement. Δs is calculated as the "sum of the distance travelled by both wheels divided by two" (iRobot Corporation, 2006). There are no details on how $\Delta \alpha$ is calculated. The manual states, "*Create* uses wheel encoders to measure distance and angle. If the wheels slip the actual distance or angle travelled may differ from *Create's* measurement" (iRobot Corporation, 2006).

To get an idea of the Δs accuracy, we collected some measurements based on varying speeds (20 mm/s – 300 mm/s) and distances (100mm – 1000mm) on a fixed straight path. The results have been summarized in Figure 3-9 and clearly show large errors at high speeds. For a full table of measurements please see the Appendix.

[9] "Real time" process priority offered by the OS scheduler, but obviously not hardware real time.
[10] Half of maximum speed: 200 mm/s.

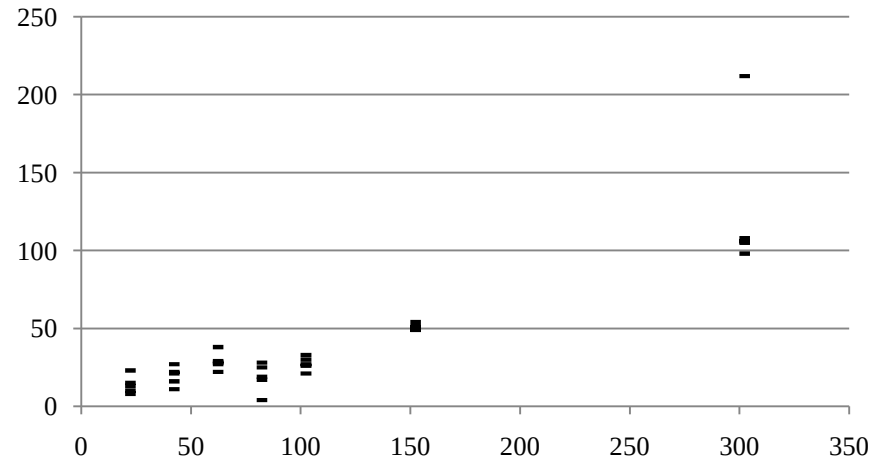


Figure 3-9: Speed / Error Relationship for Varying Distances

3.4.5 Range Finder

The mounted Sharp IR Range Finder is very precise. It can accurately measure obstacles from 100mm to 800mm away with just a few millimeters of error (Sharp, 2002). The analog signal sent by the *iRobot Create* controller ranges from 540 (obstacle near) to 75 (obstacle far) and slightly varies depending on the reflecting surface. Measurements were made to create a translation function from the analog signal to millimeters. The relationship between the two curves is shown in Figure 3-10. We chose a linear piecewise function of 20 intervals to interpolate the relationship. For a full table of measurements please see the Appendix.

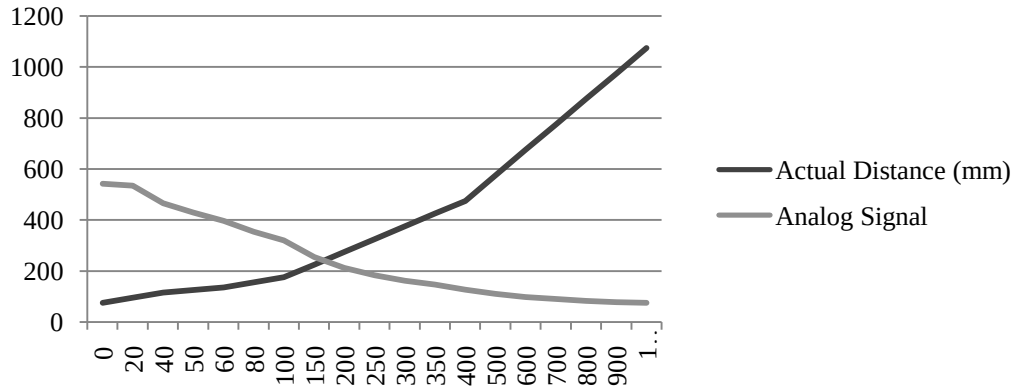


Figure 3-10: Relationship between the Range Finder Analog Signal and Actual Distance

3.5 Challenges

Having an understanding of the *emss* hardware, we continue to discuss the challenges, and therefore tasks, presented by the hardware. As with most robotics projects, one must face many challenges which are disjoint from the actual project goal or purpose. Such external challenges range from the

assembly of hardware to the communication with controllers to user interfaces for control and debugging, each an entirely different problem. Most project challenges, or core problems, rely on the successful solution for the external challenges.

The first challenge of the *emss* project is the successful assembly of the hardware components in a robust fashion. While the robot must not endure a significant amount of hardships, it must withstand minor bumps and conditions and motion stress. The mounted x86 computer must be firmly mounted in a lightweight fashion. Furthermore the different components have to be physically mounted and electronically connected without causing any conflicts in either aspect.

Once the hardware is assembled, communication with the components must be established. In *emss*, this is a two-tier challenge. First, communication between the controller (the x86 computer) and the *iRobot Create* hardware must be realized, and secondly, communication between the *iRobot Create* and external sensors must be established. The first tier is not trivial, as we must adhere to the vision of creating cross-platform software and the serial interface must be used. Furthermore, the serial interface must be abstracted to enable ease of use throughout our software. The different commands and data packets of the *iRobot Create Open Interface* must be implemented.

Being able to communicate with the *emss* hardware from software presents the next challenges: navigation and localization. One must be able to navigate the robot either by stick[11] or by algorithm, both which need to send the appropriate commands to the *Create* controller at the right time in the right interval. Furthermore, the navigation of the robot must be realized in a way which enables the possibility of localization based on the hardware's sensors. The feedback provided by the hardware should be complimented by software in order to perform more accurate localization.

After *emss* performs its given tasks, it must be able to recharge itself in order to prove its self-sustainability. The challenge has the subtasks of finding your way back to the charging station and then successfully docking itself, making contact with all four charge contacts.

Finally, in order to influence the *emss* hardware one needs an appropriate interface which complements the features of the software and hardware. The challenge here is to produce a useful cross-platform graphical user interface without hindering any of the core software.

The implications of these shortly described challenges are discussed in more detail throughout the chapters "Theory" and "Software".

[11] Meaning in the sense of a joystick where the user directly influences the movement of the robot.

4 Theory

Unaware to us at the beginning of the *emss* project, theory has become increasingly important throughout our mission. We attacked our problems with the idea to solve them programmatically, but soon realized the advantages and value of leaning back, broadening our scope, and studying theoretical solutions along with their application. This realization has opened to us a wide array of doors for exploration and research regarding robotics, and furthermore has taught us the power of applied mathematics in software. Ultimately, grappling with theoretical concepts has compensated for some of our acknowledged (and well admitted) weakness in mathematical fields such as linear algebra.

The following chapters will provide the necessary theory to understand the *emss* software framework, as well as provide some additional insight for possible solutions to problems. Because of the physical nature of our robot, specifically its constraint to the floor by both wheels and the force of gravity, we chose a 2-dimensional Cartesian space for all our theory.

4.1 Vector Geometry

A vector is formally defined as an element of a vector space. Simply put, a vector space is a set of elements with the operations of addition and scalar multiplication. This means that the elements of a vector space, vectors, can be scaled and added. A vector space is to be defined by a specific field[12], such as real numbers or complex numbers. The scalars of that vector space will be the elements of the associated field.

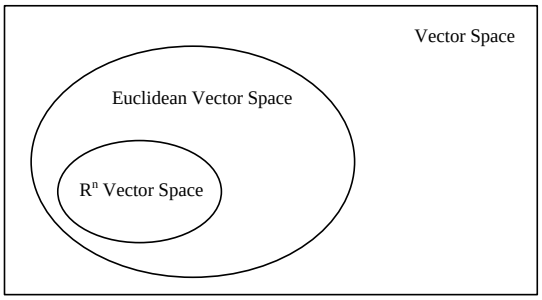


Figure 4-1: Illustration of Vector Space Subsets

A common vector space is the Euclidean vector space called $\mathbb{R}^n$, where every element is represented by a list of n real numbers. Such a vector space is a combination of a vector space and the inner product operation. Here, scalars are real numbers, addition is performed componentwise, and scalar multiplication is done on each term separately (Weisstein, 1999). The elements of a Euclidean n-Space are called Euclidean vectors: a geometrical entity with both a magnitude (length) and orientation (direction). It is mostly commonly drawn as an arrow with a starting point A and an endpoint B (Figure 4-2). The inner product operation is required by such vectors for determining the magnitude, or length, of the vector. Euclidean vectors may be added, subtracted, multiplied by a

[12] In abstract algebra, fields are algebraic structures in which the operations of addition, subtraction, multiplication, and division may be performed in a way that satisfies ordinary number.

number or another vector, and flipped around so that the orientation is reversed. These operations obey the algebraic laws of associativity, commutativity, and distributivity. A Euclidean vector in the Euclidean Space $\mathbb{R}^2$ is defined as

$$\vec{v} = \begin{pmatrix} x \\ y \end{pmatrix}$$

where the x-value describes the change on the Cartesian abscissa and the y-value the change on the Cartesian ordinate. The dot product of two vectors (also known as the inner product) results with a scalar value:

$$\vec{v}_1 \bullet \vec{v}_2 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \bullet \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = x_1 * x_2 + y_1 * y_2$$

The magnitude or length of the vector, respectively the norm of a vector, is calculated as follows:

$$|\vec{v}| = \sqrt{x^2 + y^2} = \sqrt{\vec{v} \bullet \vec{v}}$$

Many rudimentary robotics problems, such as navigation, localization, and map building revolve around geometrical states and transformations. These states can easily be described and altered using vector geometry. For example, the addition of a series of vectors may describe a specific movement through space giving its end position, as shown in Figure 4-2. By refactoring certain trigonometry calculations into vectors one can significantly enhance code readability, et cetera. Furthermore, vectors can serve a useful purpose for describing certain data structures. A two-dimensional vector, for example, could describe the current position of the robot.

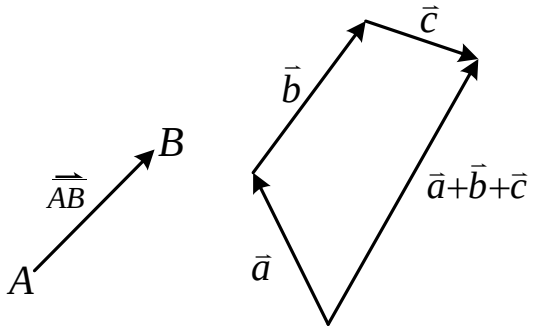


Figure 4-2: Representation of Vectors and Addition of Vectors

4.2 Transformation Matrices

Arbitrary linear transformations of Euclidean n-space vectors can be realized with $n \times n$ matrices. Such transformations include scaling, shearing, rotating, and mirroring. Figure 4-3 demonstrates how different 2×2 matrices T can transform a set of 2-space vectors.

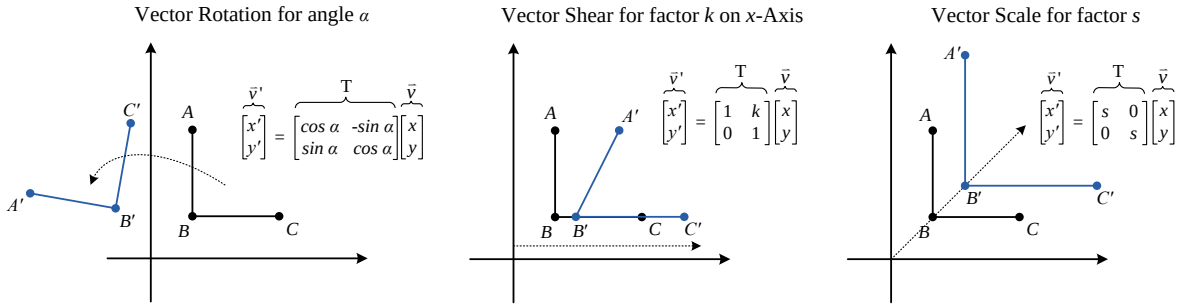


Figure 4-3: Vector Transformations

One of greatest advantages for using transformation matrices in order represent linear transformations is that transformations can be easily composed and inverted. Composition is achieved by matrix multiplication. Therefore, if T_1 and T_2 are the matrices of two linear transformations, then the effect of applying first T_1 and then T_2 to a vector $\vec{v}$ is given by the product of the two matrices[13]:

$$T_2(T_1\vec{v}) = (T_2T_1)\vec{v}$$

The ability to compose transformations by multiplying their matrices naturally allows one to undo the transformations simply by inverting their matrices. An example would be the following:

$$T = T_2T_1, \;\; T_1 = T_2^{-1}T$$

In a two-dimensional space, our robot needs only it's x and y coordinates along with the rotation (orientation) in order to describe its geometrical state[14]. If we assume our robot to be a rigid body, the needed transformations to describe its movement are translation and rotation. Restricted to our $n \times n$ transformation matrices are translations, a type of transformation known in mathematics as affine transformations. Such a transformation consists of a linear translation followed by a translation. In the terms of vector algebra, an affine transformation requires matrix multiplication followed by vector addition ($\vec{v} \rightarrow T\vec{v} + \vec{t}$). Thus, to describe our robot's movements we need a rotation matrix R as well as the translation vector $\vec{p}$. The transformation can be written as

$$M_{affine} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} = \begin{bmatrix} cos\alpha\, x_1 - sin\alpha\, x_2 + t_1 \\ sin\alpha\, x_1 + cos\alpha\, x_2 + t_2 \end{bmatrix}.$$

To realize such transformations using matrices, we must use homogenous coordinates. Using homogenous coordinates we can represent our vector $\vec{v}$ of n-dimensions as $n + 1$ dimensions, ultimately enabling us to express translations with a single matrix multiplication. We observe that the following multiplication $M_{homogenous}$ is almost identical to M_{affine}:

$$M_{homogenous} = \begin{bmatrix} \cos\alpha & -\sin\alpha & t_1 \\ \sin\alpha & \cos\alpha & t_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ 1 \end{bmatrix} = \begin{bmatrix} cos\alpha\, x_1 - sin\alpha\, x_2 + t_1 \\ sin\alpha\, x_1 + cos\alpha\, x_2 + t_2 \\ 1 \end{bmatrix}$$

[13] Note that the multiplication is done in the opposite order from the English sentence: the matrix of "T_1 followed by T_2" is T_2T_1, not T_1T_2.

[14] Of course, in order to describe its physical state, one would just add its velocity in the form of left-right wheel velocity.

Because of the nature of the homogeneous component of a coordinate vector, we can assume it will always be 1 and therefore it can be ignored. Using this technique of transformation matrices one can efficiently describe the state of a robot as well as calculate composed rotations and translations.

4.3 Curves

In general mathematics, a curve can be thought of as a set of points through which a continuously moving point passes through, creating a continuous[15] geometrical object. The need for curves can be found when trying to describe certain geometric shapes, such as a circle. Describing such a shape in the form of $\mathbb{R} \to \mathbb{R}$ is not sufficient, as the points along the "curve" of the circle shape overlap for the same input. What is required here is a function which maps $\mathbb{R} \mapsto \mathbb{R}^n$:

$$\mathbb{R} \supset I \to \mathbb{R}^n$$

where I is the interval $[a, b]$. Furthermore, in a two-dimensional space one can construct such shapes using

$$\mathbb{R} \supset I \to \mathbb{R}^2$$

$$t \mapsto \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}.$$

For example, a circle's curve C_{circle} can be easily defined as

$$C_{circle} = t \mapsto \begin{pmatrix} \cos(t) \\ \sin(t) \end{pmatrix}$$

If we look at such a curve for $I = [-\pi, \pi]$, including its x function $\cos t$ and y function $\sin t$, we see a circle shape as shown in Figure 4-4.

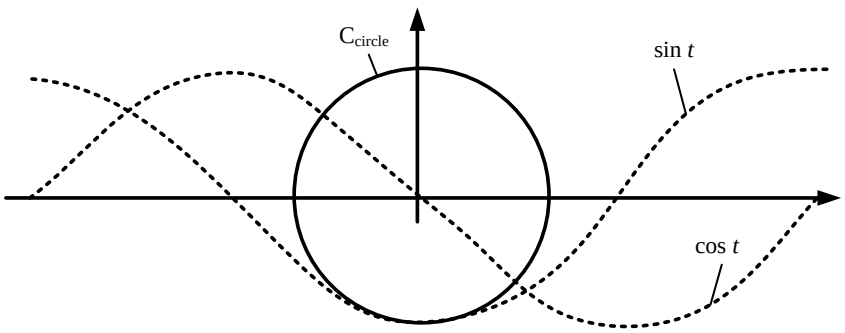


Figure 4-4: Example Circle Curve

Curves naturally exist everywhere. Where we find them especially interesting are the curves the left and right wheels of our robot leave behind as they move through space. We could even consider these curves of simple geometric shapes, such as those of a circle as shown in Figure 4-5. However there are much more powerful curves, which brings us to the next Chapter "Splines".

[15] A linearly ordered set of more than one point that is densely ordered, meaning a "line" which has no gaps.

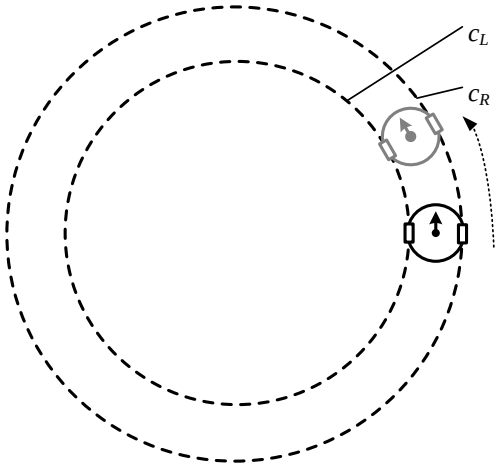


Figure 4-5: Robot Moving on Circle Curves

4.4 Splines

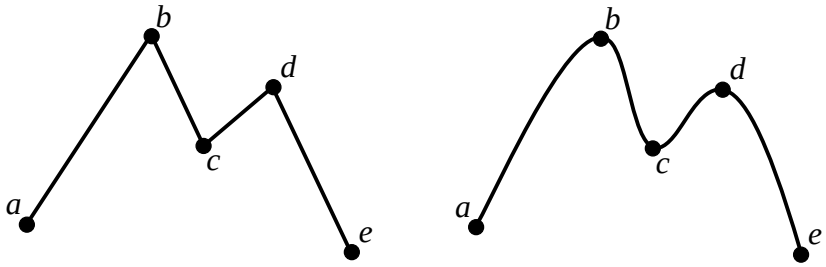


Figure 4-6: Linear Interpolation (left) and Spline Interpolation (right)

A spline is a special type of curve. In computer science, splines are most commonly known for their resulting smooth parametric curves, popular for their simplicity of construction and capacity to approximate complex shapes. In the mathematical sense, a spline is a special piecewise polynomial function which avoids problems that occur when using polynomial interpolation with polynomials of high degrees[16]. Splines present a special interest to us because of their mathematical nature to represent smooth navigational paths. For example, if one had a certain set of points to navigate through, the trivial solution is to piecewise linear interpolate a function which passes through the points. However, this causes rigid movement as you cross from one point to the next. Splines create a smooth, more natural, fluid movement through the points, as shown in Figure 4-6.

A polynomial spline $S: [a, b] \to \mathbb{R}$ consists of polynomial pieces $P_i: [t_i, t_{i+1}) \to \mathbb{R}$ where

$$a = t_0 < t_1 < \cdots < t_{n-2} < t_{n-1} = b.$$

The given n points t_i are called nodes. These define the shape of the spline, and if equidistantly distributed in the interval $[a, b]$ form a uniform spline (otherwise non-uniform).

The degree of the spline is determined by the highest degree of the polynomial pieces on the subintervals

[16] Such as Runge's phenomenon (Runge, 1901).

$$[t_i, t_{i+1}), i = 0, \dots, n-2.$$

The most simple and trivial spline has degree 0, which in turn creates nothing more than a step function. Such a curve is not continuous. A spline of degree 1 is a linear spline, and if closed forms a polygon. Such a curve is continuous, however its first derivative is not and typically represents the classical chunky robot movements. Splines get interesting at degree 3, where they form a smooth interpolation (Figure 4-7) and are continuous in both tangency and curvature.

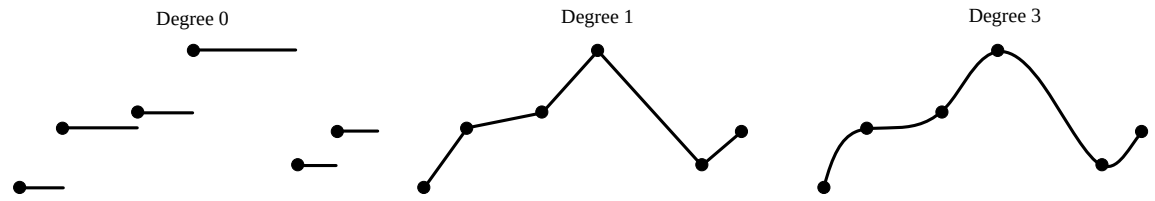


Figure 4-7: Splines with Various Degrees

4.4.1 Calculating Splines

There are several flavors of splines, amongst them B-Splines (Boor, 1978), Cubic Hermite Splines (Bartels, Beatty, & and Barsky, 1998), Catmull-Rom Splines (Catmull, 1974), and the common natural C^2 Cubic Splines to name a few. We have focused our attention on the later. The natural cubic spline is of degree 3 and must satisfy,

$$S''(a) = 0 = S''(b).$$

Here, the term natural implies that the start and end node's second derivative must equal zero. This forces the spline to be a straight line outside the interval. In each interval $[t_i, t_{i+1}]$ our spline $S(t)$ takes the form of

$$S_i(t) = a_i + b_i t + c_i t^2 + d_i t^3$$

with n intervals, $n+1$ nodes, and 4 unknowns per node totaling $4n$ unknowns. There exists $2n$ constraints by continuity and $n-1$ constraints by continuity for both $S'(t)$ and $S''(t)$, totaling $4n-2$ total constraints (Olson, 2008). Basically in order to solve a_i, b_i, c_i, d_i for each i, we examine the nodes $t_0, \dots, t_n$ with the following steps:

1. Assume $S''(t_i)$ for each i
2. Construct $S_i''(t)$ as it is linear
3. Get $S_i(t)$ by integrating $S_i''(t)$
4. Differentiate $S_i(t)$

Assuming we know $S''(t_i)$ for each i we assume

$$z_i = S''(t_i)$$

as we know $S''(t)$ is continuous. Since $S_i''(t)$ is linear and

$$S_i''(t_i) = z_i$$

$$S_i''(t_{i+1}) = z_{i+1}$$

we can write $S_i(t)$ as

$$S_i''(t) = z_i \frac{t_{i+1}-t}{t_{i+1}-t_i} + z_{i+1}\frac{t-t_i}{t_{i+1}-t_i} = \frac{z_i}{h_i}(t_{i+1}-t) + \frac{z_{i+1}}{h_i}(t-t_i)$$ (Olson, 2008)

where h_i is the interval distance between nodes. If we integrate $S_i''(t)$ twice we will get $S_i(t)$:

$$\int S_i''(t)\,dx = S_i'(t) = -\frac{z_i}{2h_i}(t_{i+1}-t)^2 + \frac{z_{i+1}}{2hi}(t-t_i)^2 + \hat{C}_i$$

$$\int S_i'(t)\,dx = S_i(t) = \frac{z_i}{6h_i}(t_{i+1}-t)^3 + \frac{z_{i+1}}{6hi}(t-t_i)^3 + \hat{C}_i t + \widehat{D}_i$$

$$= \frac{z_i}{6h_i}(t_{i+1}-t)^3 + \frac{z_{i+1}}{6hi}(t-t_i)^3 + C_i(t-t_i) + D_i(t_{i+1}-t)$$ (Olson, 2008).

Furthermore, for each interval $[t_i, t_{i+1}]$ we require $S_i(t_i) = x_i$ and $S_i(t_{i+1}) = x_{i+1}$ which imposes continuity:

$$x_i = S_i(t_i) = \frac{z_i}{6h_i}(t_{i+1}-t_i)^3 + \frac{z_{i+1}}{6hi}(t_i-t_i)^3 + C_i(t_i-t_i) + D_i(t_{i+1}-t_i)$$

$$= \frac{z_i}{6h_i}(t_{i+1}-t_i)^3 + D_i(t_{i+1}-t_i) = \frac{z_i}{6}h_i^2 + D_i h_i$$

$$D_i = \frac{x_i}{h_i} - \frac{h_i}{6}z_i$$

and

$$x_{i+1} = S_i(t_{i+1}) = \frac{z_i}{6h_i}(t_{i+1}-t_{i+1})^3 + \frac{z_{i+1}}{6hi}(t_{i+1}-t_i)^3 + C_i(t_{i+1}-t_i) + D_i(t_{i+1}-t_{i+1})$$

$$= \frac{z_{i+1}}{6hi}(t_{i+1}-t_i)^3 + C_i(t_{i+1}-t_i) = \frac{z_{i+1}}{6}h_i^2 + C_i h_i$$

$$C_i = \frac{x_{i+1}}{h_i} - \frac{h_i}{6}z_{i+1}.$$

If we plug in C_i and D_i into our $S_i(t)$ we get

$$S_i(t) = \frac{z_i}{6h_i}(t_{i+1}-t)^3 + \frac{z_{i+1}}{6hi}(t-t_i)^3 + \left(\frac{x_{i+1}}{h_i} - \frac{h_i}{6}z_{i+1}\right)(t-t_i) + \left(\frac{x_i}{h_i} - \frac{h_i}{6}z_i\right)(t_{i+1}-t).$$

We differentiate $S_i(t)$ to impose continuity on $S'(t)$ (Olson, 2008):

$$S_i'(t) = -\frac{z_i}{2h_i}(t_{i+1}-t)^2 + \frac{z_{i+1}}{2hi}(t-t_i)^2 + \frac{x_{i+1}}{h_i} - \frac{h_i}{6}z_{i+1} - \frac{x_i}{h_i} + \frac{h_i}{6}z_i.$$

Because $S_i'(t_i) = S'_{i-1}(t_i)$ we can define z_i:

$$S_i'(t_i) = -\frac{h_i}{6}z_{i+1} - \frac{h_i}{3}z_i + \underbrace{\frac{1}{h_i}(x_{i+1}-x_i)}_{b_i}$$

$$S'_{i-1}(t_i) = \frac{h_{i-1}}{6} z_{i-1} - \frac{h_{i-1}}{3} z_i + \underbrace{\frac{1}{h_{i-1}}(x_i - x_{i-1})}_{b_{i-1}}$$

$$\Rightarrow h_{i-1}\underline{z_{i-1}} + 2(h_i + h_{i-1})\underline{z_i} + h_i\underline{z_{i+1}} = 6(b_i - b_{i-1}).$$

This definition of z_i has $n-1$ unknowns and is a $(n-1) \times (n-1)$ tridiagonal system of equations:

$$\begin{bmatrix} 1 & & & & & & & \\ h_0 & u_1 & h_1 & & & & & \\ & h_1 & u_2 & h_2 & & & & \\ & & h_2 & u_3 & h_3 & & & \\ & & & \cdot & \cdot & \cdot & & \\ & & & & h_{n-3} & u_{n-2} & h_{n-2} & \\ & & & & & h_{n-2} & u_{n-1} & h_{n-1} \\ & & & & & & & 1 \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \\ \cdot \\ z_{n-2} \\ z_{(n-1)} \\ z_n \end{bmatrix} = \begin{bmatrix} 0 \\ v_1 \\ v_2 \\ v_3 \\ \cdot \\ v_{n-2} \\ v_{n-1} \\ 0 \end{bmatrix}$$

where

$$u_i = 2(h_i + h_{i-1}) \text{ and } v_i = 6(b_i - b_{i-1}).$$

4.4.2 Solving Splines Using a Tridiagonal System of Equations

A tridiagonal matrix has the property an almost *diagonal* matrix, where nonzero elements only occur in the main diagonal, the first diagonal below this, and the first diagonal above the main diagonal:

$$\begin{bmatrix} * & * & 0 & 0 \\ * & * & * & 0 \\ 0 & * & * & * \\ 0 & 0 & * & * \end{bmatrix}$$

Furthermore, tridiagonal systems of equations may be written as

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i$$

Where $a_1 = 0$ and $c_n = 0$ result in the following matrix (Conte & deBoor, 1972):

$$\begin{bmatrix} b_i & c_i & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \cdot & \\ & \cdot & \cdot & \cdot & c_{n-1} \\ 0 & & \cdot & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \cdot \\ \cdot \\ d_n \end{bmatrix}$$

These systems are interesting because they can be solved in $O(n)$ operations instead of the Gaussian $O(n^3)$ elimination. A first loop is required to eliminate all a_i and then a backwards substation produces the solution. The following code solves a tridiagonal system of equations:

```
// Fills solution into x using values in a, b, c, and d
TridiagonalSolve(array a, array b, array c, array d, array x, int n){

    // Modify the coefficients
    c[0] /= b[0];
    d[0] /= b[0];
```

```
    for(int i = 1; i < n; i++){
          double id = (b[i] - c[i-1] * a[i]);
          c[i] /= id;
          d[i] = (d[i] - d[i-1] * a[i])/id;
    }

    // Back substitution
    x[n-1] = d[n-1];
    for(i = n - 2; i >= 0; i--)
          x[i] = d[i] - c[i] * x[i+1];
}
```

Code 4-1: Solving of a Tridiagonal System of Equations

Finally, to determine a spline for a given set of nodes one must calculate h_i, b_i, u_i, v_i, then solve for z_i using a system of tridiagonal equations, and ultimately plug the values into

$$S_i(t) = \frac{z_i}{6h_i}(t_{i+1} - t)^3 + \frac{z_{i+1}}{6hi}(t - t_i)^3 + \left(\frac{x_{i+1}}{h_i} - \frac{h_i}{6}z_{i+1}\right)(t - t_i) + \left(\frac{x_i}{h_i} - \frac{h_i}{6}z_i\right)(t_{i+1} - t).$$

4.5 Controlling the Differential Steering System

A differential steering system is a design concept which many robots share. A robot with such a steering system has the advantage of being able to drive fluid curves without any mechanical steering device, such as a steering wheel. The steering is performed by creating a difference of speed between the independently powered driving wheels, enabling a wide array of different motions, including rotations around the center axis. However, controlling a differential steering system is not trivial. There are many factors to consider, and many systems which influence decisions made on how to control the steering system. In fact, one can find that many two-wheeled robots do not take advantage of a differential steering system due to its complications and restrict themselves to rotations around the center axis and straight forward movements. Furthermore, sometimes dependent systems such as localization trackers prevent a robot from performing the fluid motions of a differential system due to error induced by their organic nature.

Problems and implications set aside, we chose to research steering *emss* differentially; partly because of the attractiveness of the fluid movements produced, but more importantly because of the implicated problems produced by hard linear movements and rotations[17]. Here we describe two concepts of differential steering control, one which we consider to be of a mechanical nature and the other organic.

[17] It was observed that fixed axis rotations around the center axis would cause great error, especially in the low range of angle changes such as one or two degrees.

4.5.1 Differential Steering Using Splines

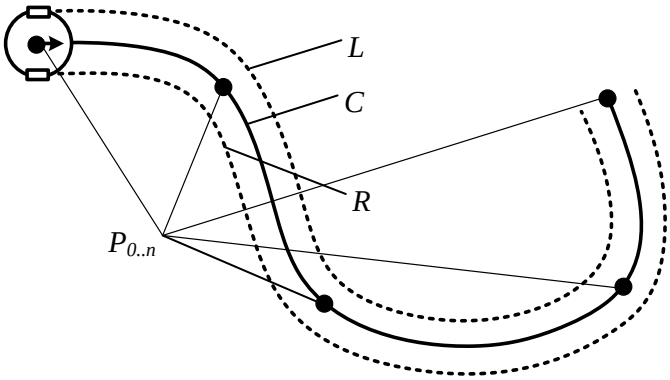


Figure 4-8: Differential Steering Using Splines

We can use spline curves to directly control the movement of our two-wheeled robot by calculating the first derivative of the spline: speed. In this scenario, each wheel must have its own spline curve to drive along, based on the given navigation points. The navigation points $P_{0..n}$ are defined by a finite series of n coordinates which the robot must visit in sequential order. Using these points we construct a spline curve C of n nodes which represents the navigational curve in a two-dimensional space. This curve is the path the center axis of the robot must pass through when traversing the points $P_{0..n}$. However, we need the curves for both the left and right wheel if we want to directly control the differential steering system. To create these curves all we need to do is traverse C and create a corresponding offset curve L and R for each wheel (Figure 4-8).

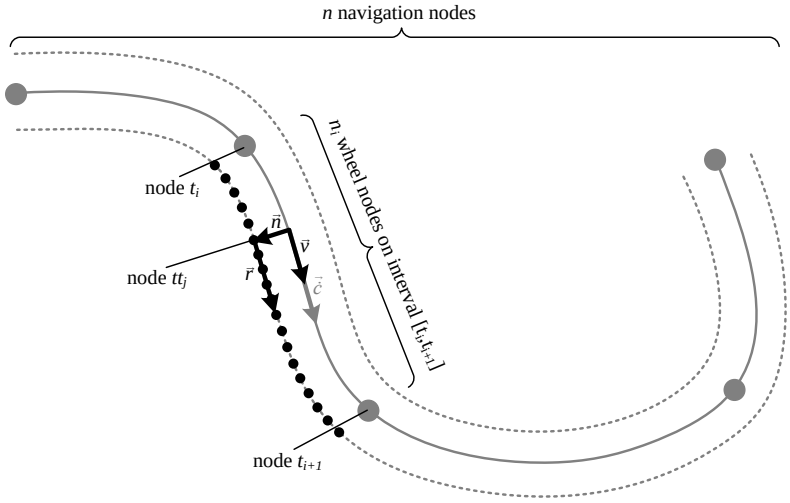


Figure 4-9: Creating Wheel Spline Nodes from a Navigation Curve

The nodes for these curves can be determined by periodically calculating the first derivative of the navigational curve C, i.e. the tangent to the curve, and shifting it in the direction of the normal vector at that point on the curve either to the left or right. If one uses uniform splines, as described in Section 4.4, care needs to be taken to either, create the wheel splines with a fixed distance between the nodes,

or, query the derivatives of the wheel splines at intervals relational to the distance between the nodes. Failing to do so will cause the yielding speeds from the wheel spline derivative to be disjoint from the navigational path. To avoid this problem without the above mentioned restrictions one should use non-uniform splines. The following steps will create the wheel splines required for the differential steering system:

First we must figure out our fixed wheel node distance s_w. As mentioned, this is important for accurately traversing the spline:

$$s_w = target\ speed * interval$$

where $interval$ is the interval at which the wheel speeds are updated on the controller. For each interval $[t_i, t_{i+1}]$ on C compute the number of wheel nodes n_i needed to satisfy the above requirements of constant distance between wheel nodes:

$$n_i = {}^{s_n}/_{s_w}$$

where s_n is the distance between our navigation nodes on the current interval $[t_i, t_{i+1}]$. The distance s_n is given by norm of the difference of the two navigational node value vectors[18]:

$$\vec{v}_i = C_i(0), \quad \vec{v}_{i+1} = C_{i+1}(0)$$

$$s_n = \|\vec{v}_i - \vec{v}_{i+1}\|$$

Compute each of the required wheel nodes tt_j for $j = 0 \dots n_i$ for the navigation curve interval $[t_i, t_{i+1}]$ as shown in Figure 4-9:

$$x_i = t_i / n_i$$

$$\vec{c} = C_i(x_i), \quad \dot{\vec{c}} = \dot{C}_i(x_i)$$

$$\vec{v} = \frac{\dot{\vec{c}}}{\|\dot{\vec{c}}\|}$$

$$\vec{n} = \begin{pmatrix} -\vec{v}_y \\ \vec{v}_x \end{pmatrix}$$

$$\vec{l} = \vec{c} + s_{offset}\vec{n}, \quad \vec{r} = \vec{c} + s_{offset}\vec{n}$$

where s_{offset} is the offset of the wheels from the center axis of the robot. Now $\vec{l}$ and $\vec{r}$ are added as points to their respective wheel curves *L* and *R*.

Once the wheel curves *L* and *R* have been created, the left and right wheel speeds for any given "tick" interval t can be extracted as so[19]:

[18] Yes, a more precise method would be to calculate the arc length of the spline on the interval.
[19] Our tick in this case would equal our wheel splines node, as they have been spaced in the distance of one tick according to our target speed.

$$v_L = \frac{\|\dot{L}_t(0)\|}{interval}, \quad v_R = \frac{\|\dot{R}_t(0)\|}{interval}$$

The following algorithm is an implementation of the above concept. It assumes mathematical structures such as a two-dimensional vector and a one-dimensional spline:

```
void calculateLeftRightWheelSpline() {

     double sw = TARGET_SPEED * INTERVAL; // Space the wheel nodes based on our
                                             wanted speed and the controller
                                             interval
     int n; // Number of navigation nodes

     // Calculate new nodes for wheel splines
     for (int ti = 0; ti < n - 1; ti++) {

           // Get the distance between the two nav nodes
           Vector2D nodePos_ti = Vector2D(navSplineX.getValue(ti, 0),
                                    navSplineY.getValue(ti, 0));
           Vector2D nodePos_tip1 = Vector2D(navSplineX.getValue(ti+1, 0),
                                    navSplineY.getValue(ti+1, 0));
           double sn = norm(nodePos_ti - nodePos_tip1);

           // Create the wheel nodes at equal distance between the two nodes
           for (int ttj = 0; ttj < (sn / sw); ttj++) {

                 // Get wheel curve node position based on navigation node
                    position
                 double xni = ttj / (sn / sw);
                 Vector2D c(navSplineX.getValue(ti,xni),
                              navSplineY.getValue(ti, xni));
                 Vector2D c1(navSplineX.getFirstDerivative(ti,xni),
                              navSplineY.getFirstDerivative(ti, xni));
                 Vector2D v = c1 / norm(c1);
                 Vector2D n = Vector2D(-v.y(), v.x());
                 Vector2D l = c + (wheelOffset * n);
                 Vector2D r = c - (wheelOffset * n);

                 wheelLeftSplineX.addNode(l.x());
                 wheelLeftSplineY.addNode(l.y());
                 wheelRightSplineX.addNode(r.x());
                 wheelRightSplineY.addNode(r.y());
           }
     }
}

Vector2D getWheelSpeed(int tick) {

     // The current node is our tick
     int node = tick;
     double t = 0.0;

     // Calculate speed
     double vl = norm(Vector2D(wheelLeftSplineX.getFirstDerivative(node,t),
                    wheelLeftSplineY.getFirstDerivative(node,t)));
```

```
    double vr = norm(Vector2D(wheelRightSplineX.getFirstDerivative(node,t),
                              wheelRightSplineY.getFirstDerivative(node,t)));
    vl *= 1.0 / INTERVAL;
    vr *= 1.0 / INTERVAL);
    return Vector2D(vl, vr);
}
```

Code 4-2: Calculation of Left and Right Wheel Splines

4.5.2 Differential Steering Using a System of Weights

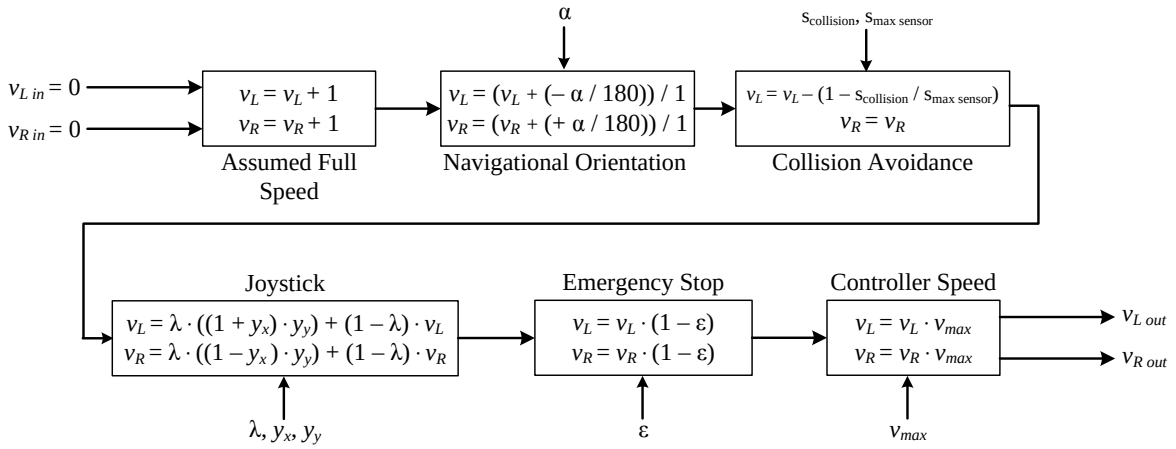


Figure 4-10: Example System of Weights

We consider splines to be a mechanical method for controlling a differential steering system. However, not in the sense of mechanical rigidness, but in the sense of their mechanical nature of automation. The start and end are given, the execution is calculated, and then faithfully realized. Once the spline is calculated all one has to do is press the button and it, the navigating, happens. When the state of the environment changes, such as an unforeseen obstacle appears, one can easily adjust the spline, either by moving a node or adding an additional node out of the way causing an arc around the obstacle. But, there are many not-so-trivial problems to solve when altering splines during the course of navigation. Most importantly, the first derivatives of the first nodes in the newly calculated splines must correspond to the state of the robot at that time. In our experience, navigational splines become exceedingly fragile with the more constraints you put on them. The curve waves tend to quickly gain energy and swing back-and-forth from node to node, ultimately creating un-navigable paths. With that in mind, we readily admit our lack of understanding of advanced spline types, and wish to spend more time researching robust but flexible splines.

An alternative to the spline approach, which we consider to be of an organic nature, is differential steering using a system of weights. Unlike the spline approach, the system of weights is completely dynamic and reacts solely on the current perception on the state of the environment at that moment of time. Thus we call it organic, because it constantly adapts to its surrounding environment. In a sense, it is not interested how it will exactly reach its goal, but rather how it will at that moment of time in that current state strive towards its goal.

Essentially, the system revolves around two variables; the left and right wheel speed v_L and v_R. These create a channel $W[v_L, v_R]$ in which different weights can be independently attached to, defined as

$$W(\vec{v}) : w_{i-1}(\vec{v}) \cdot w_i(\vec{v}),\ \ i = 1..n$$

where n is the number of weights in the channel, $\vec{v} = \binom{v_L}{v_R}$, and $w_{0..n}$ are the functions which define the weights.

Such weights influence the variables v_L and v_R by altering them based on external inputs such as sensors or switch bits. On the channels one end, raw zero-speeds $v_{L\,in}, v_{L\,in}$ come in, which are then altered by the system of weights, and on the other end the final speeds $v_{L\,out}, v_{L\,out}$ come out, which in turn are sent to the controller. Each weight in the system works independently from others, using only the channel inputs or its own inputs. A more complex system can be created by adding additional variables to the channel, such as a time tick.

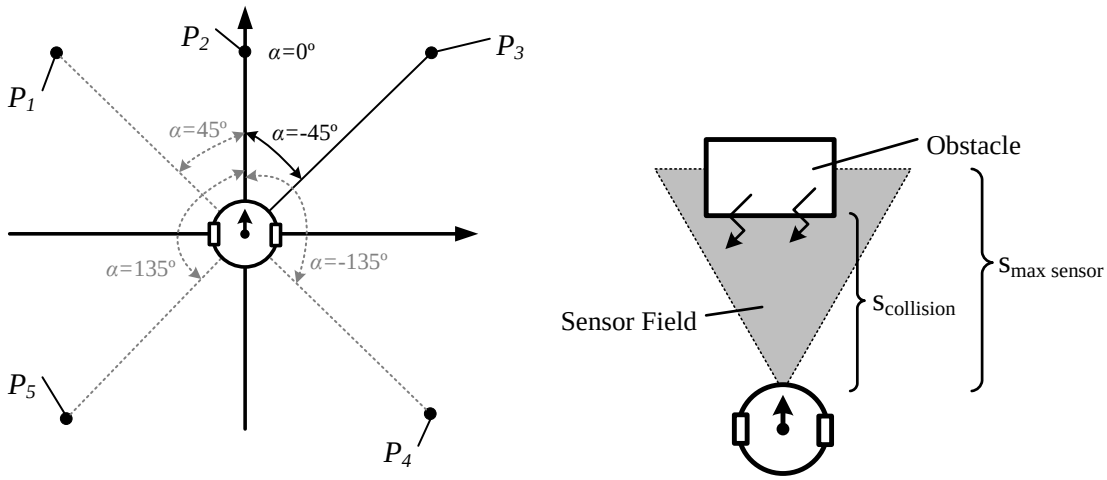


Figure 4-11: Navigational Orientation Input (left), Collision Avoidance Inputs (right)

While we have not yet spent a sufficient amount of time on this concept, and neither implemented such a system, we plan to do so in the future and present the following draft system (depicted in Figure 4-10) consisting of six weights. The system keeps the variables between $[0; 1]$ until the last weight, where it is multiplied by a factor suitable for the controller. Our first weight in W assumes that we just want to drive with full speed:

$$v_L = v_L + 1$$

$$v_R = v_R + 1$$

Immediately after, the system will drive towards the next navigation point by orienting towards its direction. This weight requires the input α which is defined the angle between the robots orientation and the next navigation point, as shown in Figure 4-11 and is defined as

$$\alpha = (-180,180]$$

$$v_L = \frac{v_L + \left(\frac{-\alpha}{180}\right)}{1}$$

$$v_R = \frac{v_R + \left(\frac{+\alpha}{180}\right)}{1}$$

This results in a steering effect towards the general direction of the navigation point. If the navigation point is straight ahead, α is 0 and thus there is no difference between v_L and v_R. Furthermore, if the navigation point is straight behind, α is 180 and therefore $v_L = 0$ and $v_R = 1$.

The next weight in the channel avoids collisions with obstacles ahead by steering to the left and requires sensors inputs $s_{collision}$ and $s_{\max\,\text{sensor}}$, shown in Figure 4-11:

$$v_L = v_L - \left(1 - \frac{s_{collision}}{s_{\max\,\text{sensor}}}\right)$$

$$v_R = v_R$$

The closer the collision, the harder the turn to the left, thus avoiding the obstacle quicker. If there is no obstacle ahead, then $s_{collision} = s_{\max\,\text{sensor}}$.

The joystick weight allows a joystick to intercept normal movements towards the navigation point and control the robot. The input λ defines whether the joystick is on (1) or off (0) while the values y_x, y_y represent the yoke values of the joystick where y_x is the left / right movement and y_y is the forwards / backwards movement:

$$\lambda = 0|1,\ \ y_x = [-1;1],\ \ y_y = [-1,1]$$

$$v_L = \lambda \cdot \left((1 + y_x) \cdot y_y\right) + (1 - \lambda) \cdot v_L$$

$$v_R = \lambda \cdot \left((1 - y_x) \cdot y_y\right) + (1 - \lambda) \cdot v_L$$

The emergency stop weight allows a kill bit ε to stop any movement of the robot.

$$\varepsilon = 0|1$$

$$v_L = v_L(1 - \varepsilon)$$

$$v_R = v_R(1 - \varepsilon)$$

Finally, the controller speed weight adjusts v_L and v_R to the input speed $v_{\max}$:

$$v_L = v_L \cdot v_{\max}$$

$$v_R = v_R \cdot v_{\max}$$

The weights for v_L and v_R can be summarized in single statement as such:

$$v_L = \left(\left(\lambda \cdot \left((1 + y_x) \cdot y_y\right) + (1 - \lambda)\right) \cdot \left(\frac{1 + \left(\frac{-\alpha}{180}\right)}{1} - \left(1 - \frac{s_{collision}}{s_{\max\,\text{sensor}}}\right)\right)\right) \cdot (1 - \varepsilon) \cdot v_{\max}$$

$$v_R = \left(\lambda \cdot \left((1 - y_x) \cdot y_y\right) + (1 - \lambda) \cdot \left(\frac{1 + \left(\frac{+\alpha}{180}\right)}{1}\right)\right) \cdot (1 - \varepsilon) \cdot v_{\max}$$

4.6 Emulating Differential Steering System Movement

Emulating the robots movement through space over time can serve different purposes. Whether it is for simulation purposes or for localization of the robot, et cetera, it is typically an important component for any robotic software. Furthermore being able to emulate the robots movement and possibly even typical deviations from the commanded movement proves to be useful for development and testing of different navigational concepts and algorithms. In hindsight, we wish we would have developed a robust emulator from the start, but are pleased with our final implementation and hope to profit from it in future development.

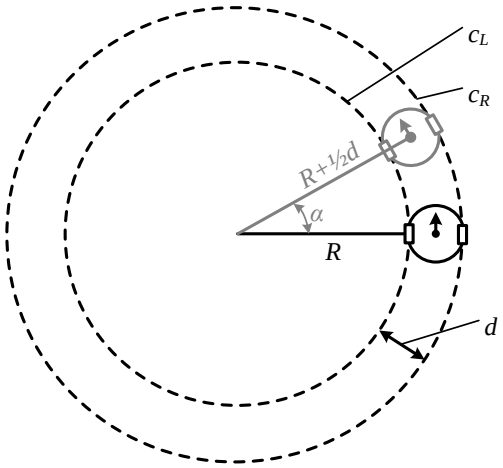


Figure 4-12: Robot Moving CCW on Circle Arcs

The tackled problem here is that the *emss* robot has a differential steering system with independent wheel velocities $v_L$and v_R. Our approach is based on the movement of each wheel as arcs of circles (Krusi S. , 2008) as shown in Figure 4-12. The radius of these circular trajectories is given by the difference of velocities. With this approach one immediately assumes $v_L \neq v_R$, as when $v_L = v_R$ the radius $R = \infty$ and the emulation for $v_L = v_R$ is trivial. We know v_L and v_R as well as the diameter d between the two wheels. Furthermore τ is the period. Our $c_L$and c_R represent the inner and outer circumferences on which the wheels traverse with velocity v_L and v_R. Our goal is to get an expression relating the radius *R* of the robot's circular trajectory in terms of the velocities. Let

$$c_L = v_L \cdot \tau = 2\pi R \qquad (1)$$

$$c_R = v_R \cdot \tau = 2\pi(R + d) \qquad (2)$$

Subtracting the circumferences and velocities lets us solve for τ (v_L and v_R are in $^m/_s$):

$$c_R - c_L = 2\pi d \qquad (3)$$

$$v_R - v_L = \frac{c_R - c_L}{\tau} \qquad (4)$$

$$\tau = \frac{2\pi d}{v_R - v_L} \qquad (5)$$

Combining equations 1 and 5 and plug in for τ gives us

$$2\pi R = v_L\left(\frac{2\pi d}{v_R - v_L}\right) \qquad (6)$$

$$R = \frac{v_L d}{v_R - v_L} \qquad (7)$$

Equation 7 lets us calculate the traversed circle radius R with our known values. Consequently, by measuring the distance S_L travelled by the arbitrarily chosen left wheel (arc length of c_L), we can extract the change in angle α over time t as

$$\alpha = \frac{S_L}{R}$$

where $S_L = v_L \cdot t$. The offset of the robot over time t can trivially be calculated using $R + \frac{1}{2}d$ and α.

4.7 Height-Map Based discovery

There are many different approaches for navigating in an unknown area, a common problem in robotics. A trivial method is to drive straight forward as long as there is no collision. Upon reaching a collision, turn a fixed arbitrary direction until there is no collision ahead and continue, peering slightly to the negated direction in order to "stick to walls". But this idea has some conceptual faults. First, it is mostly not possible to explore the entire unexplored area because the robot will follow the outer perimeter in an infinite loop. This brings us to the next problem; when should the exploration be finished?

Our approach for exploring unknown areas is a technique based on height maps. Theses maps represent the environment in a three-dimensional structure – two for the position and one we label as the height. One can picture the height map as the surface of a terrain were unknown area is defined as high and discovered space is low. The further one looks, the higher the map becomes as the area is "more" unknown. Furthermore, a complementary environment map keeps track of what is an *obstacle*, *discovered*, or *unknown*.

The first generation of the height map is entirely marked as *unknown height*. The algorithm floods the height map starting at the current position of the robot. Expanding outwards in all directions, the height of each cell of the map is consequently increased by 1 as long as that corresponding cell is *unknown* in the environment map. This creates a surface with downward pinches in it anywhere where the status map is marked as *obstacle* or *discovered*.

```
heightMap[][] = int[mapWidth][ mapHeight];
status[][] = int[mapWidth][ mapHeight];
peak = (x=0,y=0,height=0);

// Fill the height map with unknown height values
for(x := 0 to mapWidth)
     for (y := 0 to mapHeight)
           heightMap[x][y] := TERRAIN_UNKNOWN_HEIGHT;

// Add the actual robot position
queue.enqueue(Point(x,y,0));

// Generate the height map
while (is not queue.empty()) do
```

```
    p := queue.dequeue();
    h := p.height + 1;
    for (each direction)
          if (heightMap[p.x][p.y] == TERRAIN_UNKNOWN_HEIGHT
                or heightMap[p.x][p.y] > h)
                switch (status[x][y])
                      case Obstacle
                            heights[x][y] := 0;
                            break;
                      case Discovered
                            heights[x][y] := 0;
                            break;
                      case Unknown
                            heights[x][y] := h;
                            queue.enqueue(Point(x,y,h));
                            if (peak.height < h)
                                  peak.x := x;
                                  peak.y := y;
                                  peak.height := h;
end
```

Code 4-3: Height Map Construction

The algorithm starts by navigating the robot to the highest point on the height map, known as the *peak*. If this point is reached the map is re-calculated as in Code 4-3 and the robot moves to the next *peak*. The algorithm is based on the idea that when a human explores an unknown house, he does not immediately search each room in detail. Rather, he goes from room to room to make an overall view about the house then he goes to every room again to fill in the details of his exploration. This algorithm uses the same principle. Because of the nature of how the height map is calculated, i.e. distant unknown places are very high, the larger an unknown area the greater height it will accumulate.

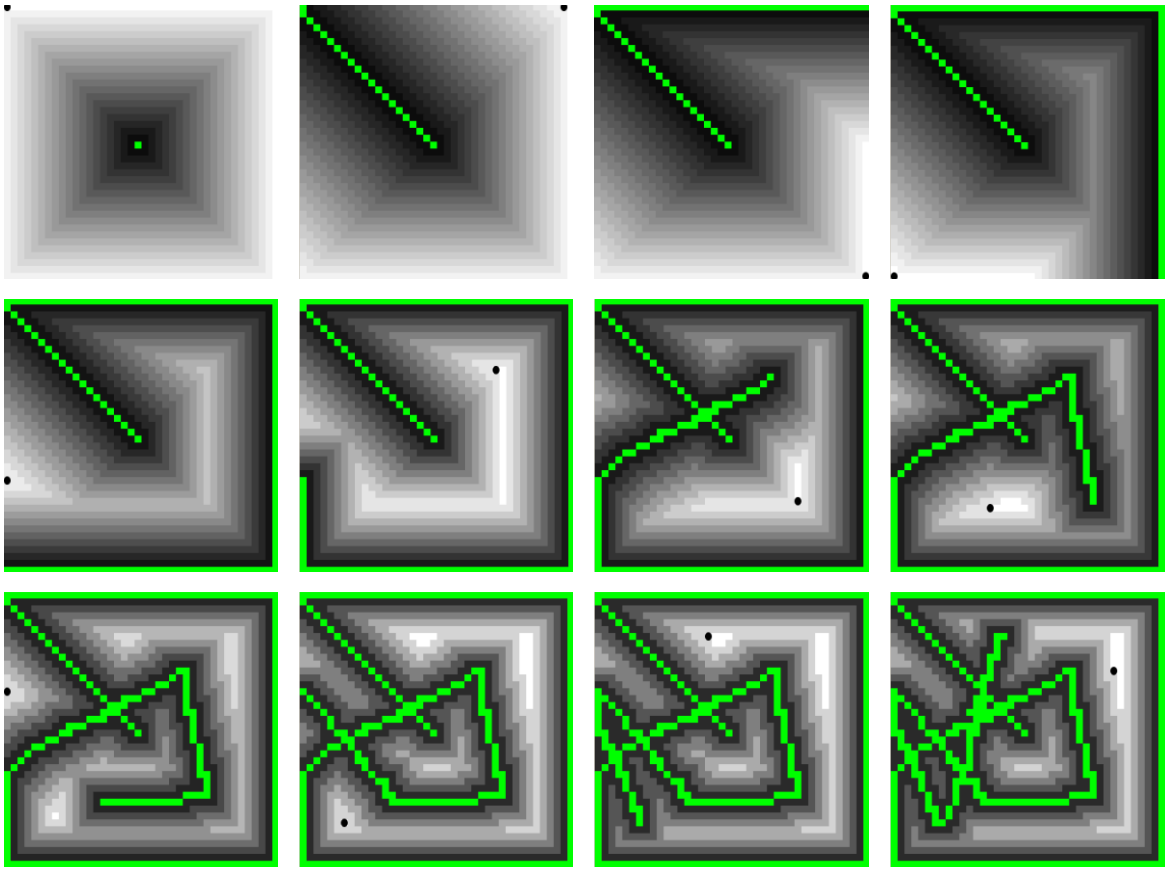


Figure 4-13: Height Map Navigation Along Peaks

The main disadvantage of this algorithm is that the highest points jump from one side of the map to the other, alternating between the largest unknown areas. This results in very impractical navigational routines, especially in larger areas. This scenario is shown in Figure 4-13 where the round black point represents the highest point in the map and the robots route is painted in green.

Therefore an extension to the algorithm is presented using cut levels. The idea of using a cut level is that we start in an unknown environment where all space around the robot is unknown. Instead of moving to the highest point, which theoretically is infinitely distant, the robot should restrict itself to a certain height, driving along the boundary this restriction creates. One can visualize the cut level as a horizontal intersection in our height map, where raising and lowering the cut level will increase and decrease the surface to explore.

However, there are still two open-ended problems. One is to decide at which point the exploration algorithm should be finished, while the other is introduced by the cut level extension: how to drive along the edge of the cut level. The first problem, when to terminate the algorithm, has various answers. It could be if all heights are zero; that means the entire area is explored and categorized as *obstacle* or *discovered*. Alternatively, one could define a certain cut level and if there are no heights over this value the algorithm is finished. The later is more efficient and timesaving because if a very low cut level is reached the cut level curve to drive along the edge is very chaotic. Driving along the edge of the cut level can be solved by sorting the cut level edge points in a specific manner. The sorting algorithm is defined as follows:

```
//sort the cut points
for (i := 0 to listOfCutPoints.size())
     distance = infinity;
     nearestPoint = 0;
     for(k := i+1 to listOfCutPoints.size())
            if(distance between listOfCutPoint[i] and listOfCutPoint[i]
                  > distance)
            distance := distance between listOfCutPoint[i] and
                  listOfCutPoint[i];
            nearastPoint := k;
     if(i+1 not listOfCutPoints.size()))
            swap(listOfCutPoint[i+1], listOfCutPoint[nearestPoint]);
```

Code 4-4: Sort Algorithm for Cut Level Edge Points

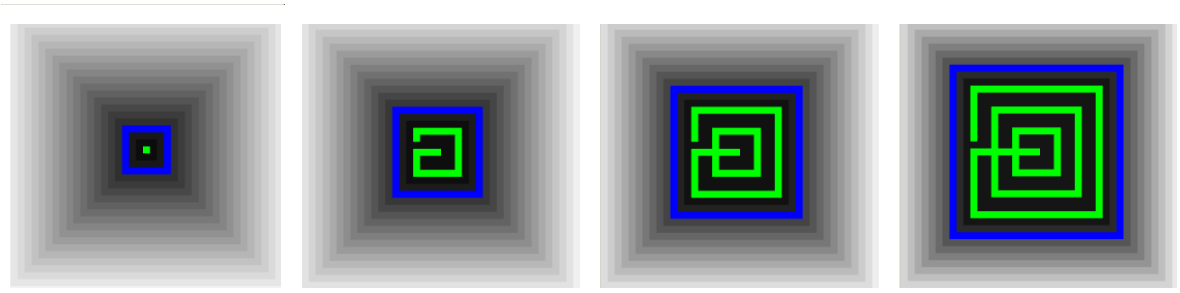


Figure 4-14: Navigation algorithm using cut points without obstacles

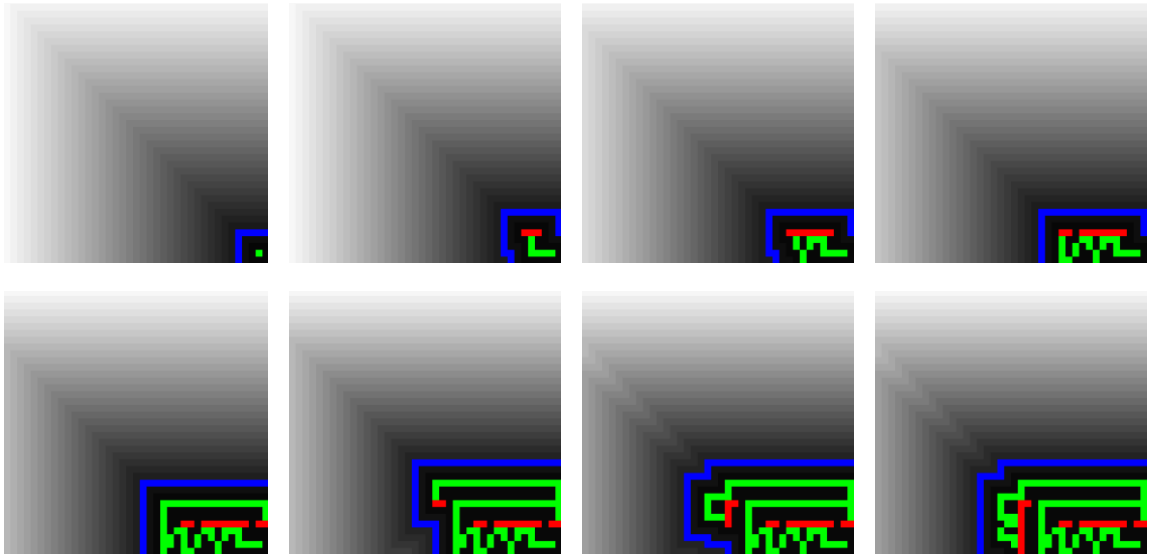


Figure 4-15: Navigation algorithm using cut points with obstacles

Figure 4-14 visualizes how the algorithm with the cut points works. The cut points are painted in blue, while the robots path is in green. This is the optimal case when no obstacles are detected. However, most spaces contain obstacles; at the very least the wall which limits the area. Figure 4-15 shows a simulation of a space with an obstacle (red). When the robot detects an obstacle in his way, the height map is re-calculated, ultimately changing the navigational behavior.

In conclusion, in order to realize such an algorithm, one must limit the height map to certain area, as our finite hardware forces us to do so. At the beginning the entire area is unknown, which means no information about the size is available. To avoid the problem of infinite maps, we use a fixed-size local height map where the robot is in the center. This local height map moves with the robot, whereas the environment map is fixed, as shown in Figure 4-16.

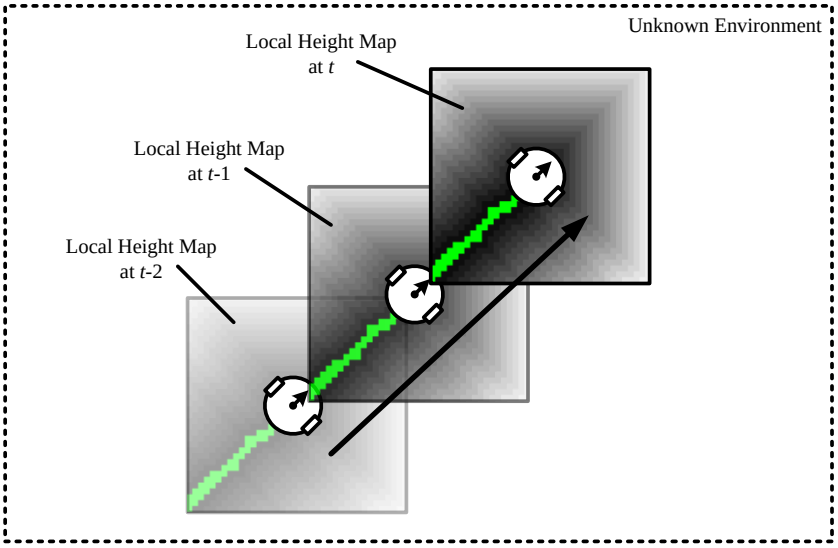


Figure 4-16: Local Height Map Concept

5 Software

While the *emss* project has a significant amount of invested energy in solving hardware problems, the focus is software. From a computer science enthusiast's point of view, one must first bite through the painful electrical engineering hardware related problems before finally tackling software issues and writing code. The software related problems of *emss* can be categorized into three components: communicating with hardware, solving the problems of objectives, and presenting the state of the software to the user. While these categories can be logically stacked on-top of each other, starting with hardware communication, the implementation process of our software fashioned a mix of all three, gradually improving on each category.

In software engineering, the traditional school of thought practices a one-way, three-tier, model consisting of a data tier, application tier, and presentation tier. While we adhere to these software principals, we have deliberately chosen not to religiously follow them. In the field of robotics it is much more profitable to focus on the componentization of different modules which naturally serve different tasks. This was aware to us from the start of the project, and in turn we largely profited from creating a framework of "hot-swappable" components.

Furthermore, it is worthwhile noting that the current software framework described here has undergone an evolutionary process. Before the start of the implementation process, the design phase provided us with an original structure based on our requirements. As requirements changed due to the research nature of the project, and certain parts of the structure proved no longer suitable, we scratched those design choices and made new ones. Changing our original software structure has cost us many long nights, but ultimately improved the overall design. All said and done, we would like to acknowledge that despite our enthusiasm to create a sound software design and structure, there are without doubt areas for improvement. The heavy mix of theory and implementation has made us jump through some high hoops.

5.1 Environment

Because of the natural distancing progression of software languages and operating systems, the choice and understanding of the environment of software has become increasingly important. Working with microcontrollers in the field of robotics, we chose the programming language of *C++*, which is well proven in the industry as a robust middle level language[20]. Because *C++* by itself does not offer any support in creating cross-platform graphical user interfaces, we use the *Qt* framework (version 4.4) from *Trolltech* to assist us. We are very proud to be able to offer a software framework in which every component, including components such as serial communications, cross-compile on Windows, Linux, and OS X. All software components are written in C++ and are fully compatible with any of the GNU 3.0 compilers. Additionally, we have made use of binary libraries. The core components of the *emss* software are packaged into a library, which in turn is linked to by any of the user interfaces. On Windows, the Core library is dynamically linked, while on Unix it is statically linked.

[20] Middle-level languages compromise a combination of high-level features and low-level advantages.

5.2 Software Model

The software which essentially runs our *emss* robot is divided into different components which ultimately make up the *emss* Core, described in detail in Section 5.3. Interfacing applications, which create an instance of the Core and provide the necessary connections for user interaction, have been omitted from our model as they are irrelevant to the functionality of the framework.

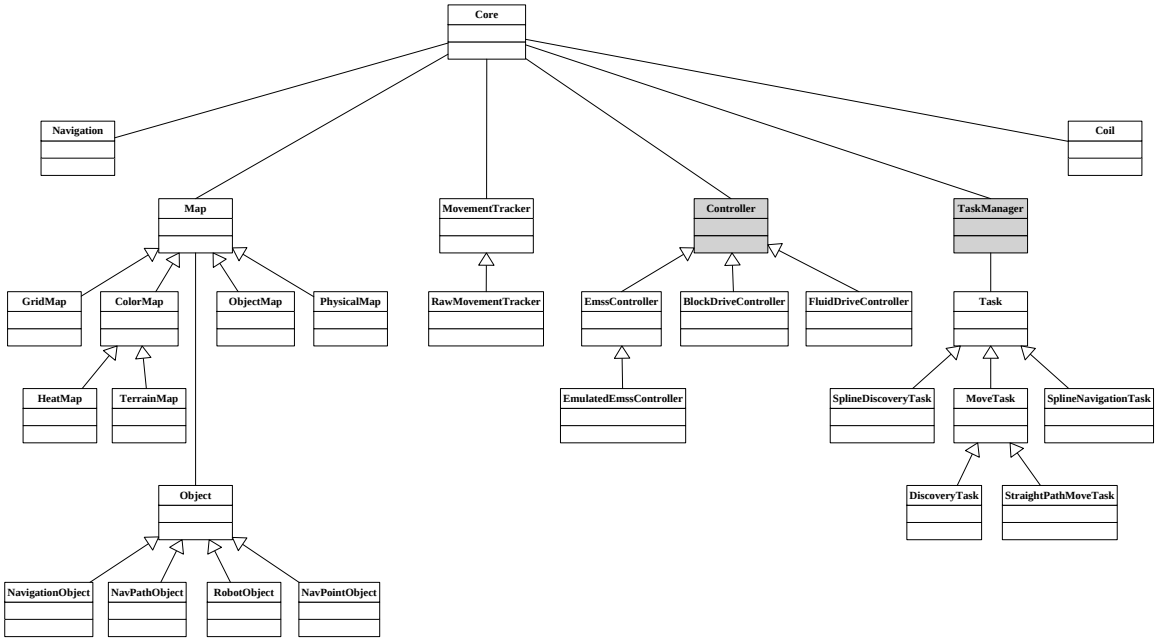


Figure 5-1: *emss* Core Domain Model

Figure 5-1 represents the final *emss* Core class structure. The Core class contains all the components, namely Navigation, Map, Movement Tracker, Controller, Task Manager, and COIL, and is responsible for instantiating them correctly in memory as well as de-allocating them. There are two running threads in the Core: Controller and Task Manager. The Task Manager is responsible for scheduling and executing Tasks, while the Controller is responsible for consequently communicating with the *emss* hardware. The design of separating Task execution and Controller execution produces a more robust controller, allowing certain logics such as cliff monitoring and emergency stops to be executed in a guaranteed fashion, even if a Task locks up and is no longer responding.

Our design has high transparency, allowing every component to "see" every other component, with one exception of COIL, which must only be accessed by the Controller. There are several lines of communication in the form of slots and signals[21], the most important between the Controller and Movement Tracker, and between a Task and the Controller. In our design, all forms of action other than communication with hardware are realized through Tasks. While the Task Manager allows more than one instance of Task, only a single Task can be executed at once – part of the design to prevent

[21] Here we took advantage of *Qt's* slot and signal design. If you are unfamiliar with this design concept, we recommend *Qt's* introductory document found on their website at http://doc.trolltech.com/4.4/signalsandslots.html.

multiple influences on the hardware communication interface. The Map and Movement Tracker components make up our data tier, assisting tasks in their job. Neatly packaging a set of functionally, the Navigation class also assists Tasks by providing navigational data structures such as splines and navigation points (way points).

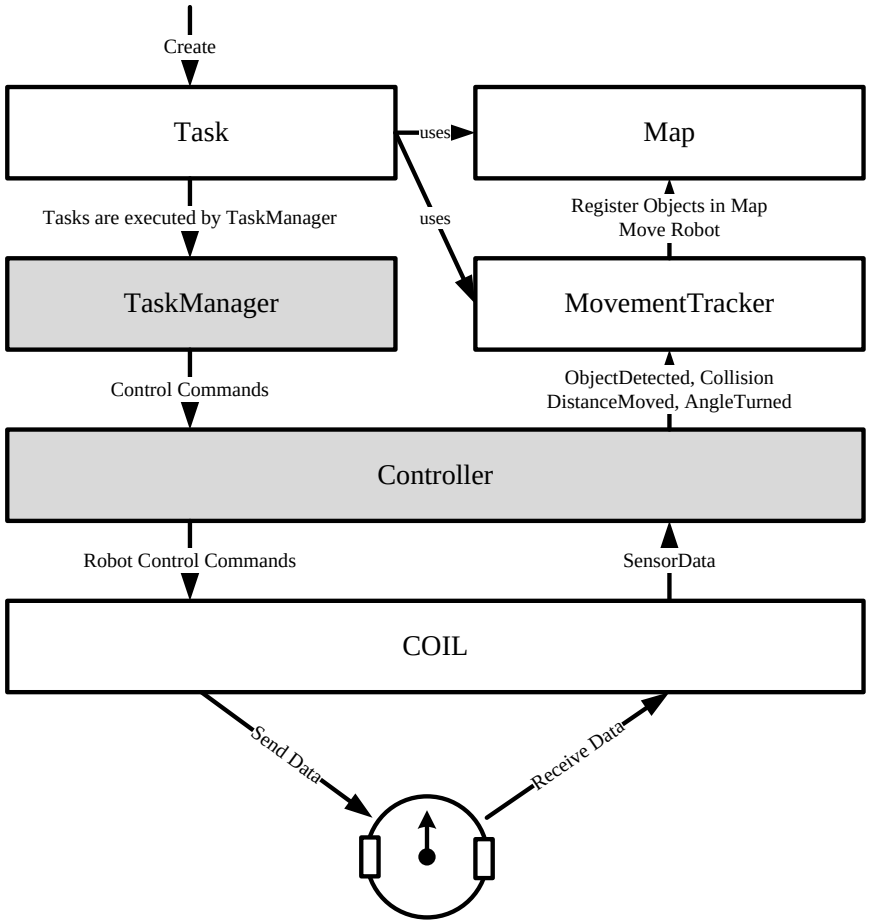


Figure 5-2: *emss* Basic Lines of Communication

The model in Figure 5-2 shows the basic lines of communication in the *emss* Core. When a Task is created, and eventually executed, it uses the different Maps and the Movement Tracker as its data source. The Movement Tracker holds the current position and orientation of the robot, while the Maps contain information about the environment, such as obstacles or previous collisions. The Task Manager is responsible for executing the Tasks one by one. The Controller receives the control commands from the Tasks and forwards them to COIL. COIL produces a data packet and sends the command over a serial port to the *iRobot Create* controller. The robot interprets the commands and executes them directly on the hardware. Per request of the Controller, COIL reads sensor data from the *iRobot Create* controller, translates them, and forwards them back to the Controller. The Controller in turn interprets the sensor data and passes them along to the Movement Tracker and appropriate Maps. For further understanding how the Core components work with each other, we present two scenarios in Section 5.2.1 and 5.2.2.

5.2.1 Fluid Drive Controller Scenario

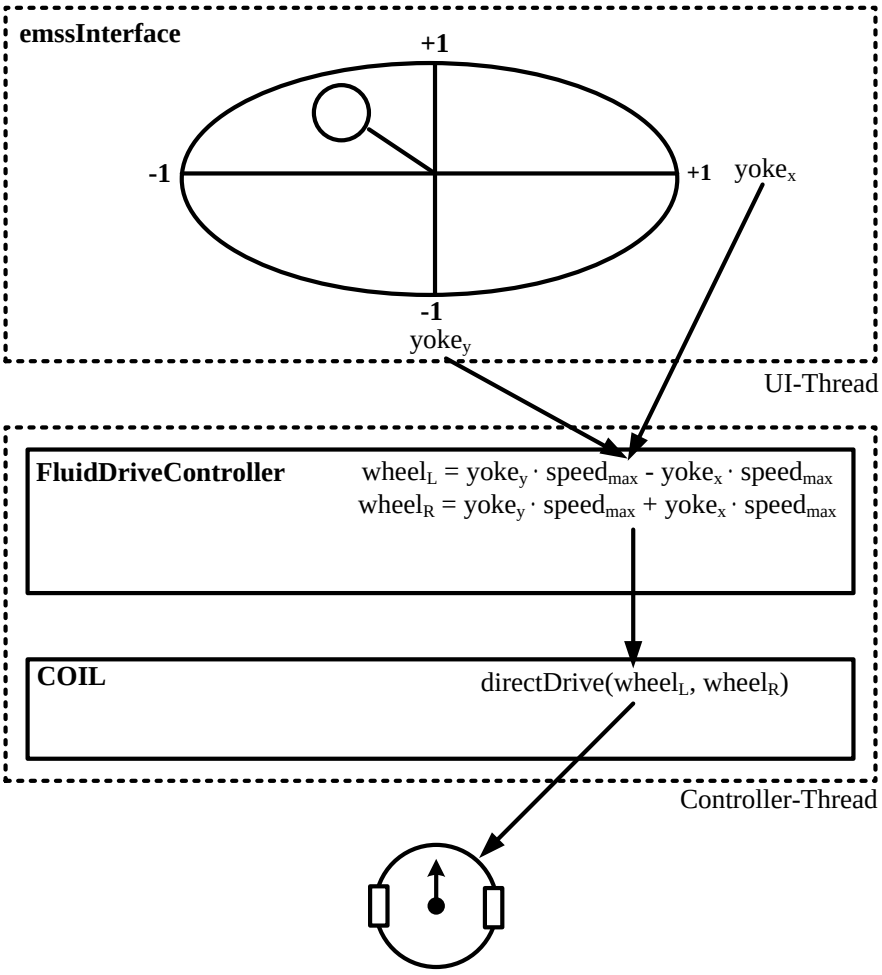


Figure 5-3: Fluid Drive Controller Scenario

The Fluid Drive Controller is a simple Core Controller which can control the robots movement by joystick. Its details can be found in Section 5.3.3.2. The *emss Interface* includes a joystick widget which sends the position of the *x*-yoke and the *y*-yoke upon change to the Fluid Drive Controller. In turn the Controller continuously calculates the left and right wheel speeds, based on the yoke positions, and forwards them to COIL, ultimately sending the movement commands to the *iRobot Create* controller.

5.2.2 Emss Controller Scenario

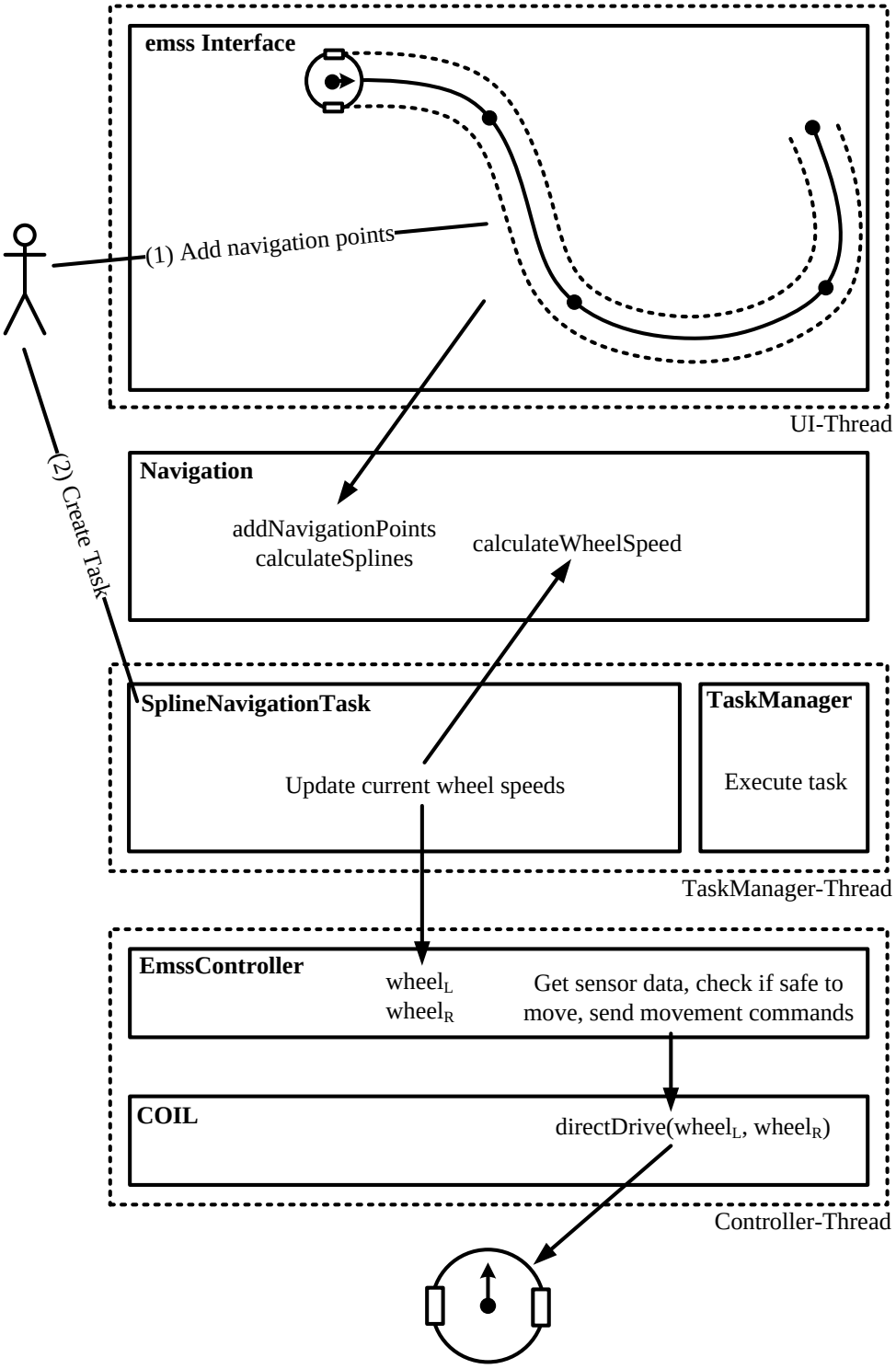


Figure 5-4: Emss Controller Scenario

The Emss Controller scenario, visualized in Figure 5-4, uses the Navigation component and is slightly more complex. First an actor must set navigation points where the robot should pass through. This could come from either the graphical user interface or another Task. When these points are added, the Navigation component stores them and automatically creates the correlating splines for each wheel.

Furthermore, a Spline Navigation Task, which is executed by the Task Manager, will extract the needed wheel speeds from the Navigation splines and forward them to the Controller. Finally, the Emss Controller may "choose" to send the movement commands through COIL to the robot hardware, given that the action is deemed safe, et cetera.

5.3 Core Components

Any application which interfaces with *emss* must initialize the Core object. The allocation and initialization of the Core class just allocates the data tier, in turn creating an empty environment and unknown state of the world, disconnected from the hardware. To use the *emss* Core and communicate with hardware, the application must call *connect(controller,tracker,port)* with the desired parameters on the Core object, which in turn creates all the connections between software and hardware, as well as lines of communications between the appropriate components. When finished, the application is responsible for calling *disconnect()* and deleting the Core object.

```
// Initialze Core and connect on COM4 using Block Drive Controller and Raw
// Movement Tracker
Core *core = new Core();
core->connect("Block Drive", "Raw", "COM4", false);
// Core is now connected and ready to be used...
core->disconnect();
delete core;
```

Code 5-1: Simple Usage of Core Class Example

5.3.1 Library

The Library package is not considered a Core component, but however is an important part of the *emss* Core as it provides a substantial amount of functionality. It is a collection of classes which are used by many different components, such as math structures or debugging routines. Some trivial Library classes such as SleeperThread, Util, and TerrainPoint are not discussed in further detail.

5.3.1.1 Vector2T

Vector2T is an abstraction of a two-dimensional vector. It realizes most of the mathematical operations operable on vectors, such as addition, scalar multiplication, dot products, et cetera. Vector2T is a template class supporting different internal data types such as doubles and floats, and has three defined typedefs of Vector2D (double), Vector2F (float), and Vector2L (long double).

```
// Basic vector operations and computations
Vector2D v1(0.5,0.8);
Vector2D v2(4.0,2.2);
Vector2D v3 = v1 + (v2*2);
double distance = Vector2D::dist(v1,v2);
double angle = Vector2D::angle(v1,v2);
```

Code 5-2: Basic Vector2T Operations and Computations Example

The original Vector2T class was implemented by our advisor Prof. Dr. Joachim Wirth.

5.3.1.2 Trafo2T

Two-dimensional transformation matrices such as rotation matrices can be realized using the Trafo2T template class. Just as the Vector2T class, mathematical operations on transformation matrices have been implemented. There are three defined typedefs of Trafo2D (double), Trafo2F (float), and Trafo2L (long double). Multiple transformations can be appended by using the * operator.

```
// Basic rotation and translation
Trafo2D t1 = Trafo2D::trans(6.0,5.0) * Trafo2D::rot(Rad(45));
Trafo2D t2 = Trafo2D::trans(1.0,1.0);
Trafo2D t3 = t2 * t1;
```

Code 5-3: Basic Trafo2T Rotation and Translation Example

The original Trafo2T class was implemented by our advisor Prof. Dr. Joachim Wirth.

5.3.1.3 Spline1T

This template class implements different one-dimensional spline approaches. As discussed in section 4.4, splines are smooth and efficient curve interpolations of a finite set of nodes. The different kernels we implemented include SplineKernalBruenner, SplineKernelNRNatural, and SplineKernelStoer. Nodes can be freely added and removed from the spline. When a call to *getValue(node,t)* is made, Spline1T automatically decides whether or not an internal recalculation of the spline coefficients is needed, maximizing the ease of use to the user. Each kernel has its own implementation of *getValue(node,t)*, *getFirstDerivative(node,t)*, and *getSecondDerivative(node,t)*. Two dimensional curves can easily be realized by using two instances of Spline1T, as mentioned in Section 4.3. There are three defined typedefs of Spline1D (double), Spline1F (float), and Spline1L (long double).

```
// Create two splines, one for each dimension and add some nodes
Spline1D Sx = Spline1D(SplineKernelNRNatural);
Spline1D Sy = Spline1D(SplineKernelNRNatural);
Sx.addNode(0.0); Sy.addNode(0.0);
Sx.addNode(1.4); Sy.addNode(1.1);
Sx.addNode(3.1); Sy.addNode(2.0);
Sx.addNode(6.7); Sy.addNode(5.4);
Sx.addNode(8.0); Sy.addNode(1.1);

// Draw each spline segment with fixed number of dots
int DOTS_PER_SEGMENT = 20;
for (int n = 0; n < Sx.getNodeCount() - n; n++) {
     for (int t = 1; t < DOTS_PER_SEGMENT; t++) {
            double px = Sx.getValue(n, (double)t / (double)DOTS_PER_SEGMENT);
            double py = Sy.getValue(n, (double)t / (double)DOTS_PER_SEGMENT);
            graph.drawPoint(px, py);
     }
}
```

Code 5-4: Example Usage of Spline1T Class – Drawing a Curve

5.3.1.4 Debug

The Debug class offers some debugging assistance, especially during runtime, by encapsulating common console commands. Printing color-formatted sets of values along with an automatic

timestamp can be very easily achieved. Additionally, calling *setOutput(widget)* allows an application to re-route all debugging information to any given text widget.

```
// Re-route debugging information to our widget
Debug::setOutput(consoleWidget);

// Simple debug messages
Debug::print("The shorthand value of PI is %1, while e is %2", 3.14, 2.72);
Debug::warning("This is a warning");
```

Code 5-5: Example Debug Class Utilization

5.3.2 COIL

The *Create Open Interface Library*[22], or COIL, was originally implemented by Jesse DeGuire and Nathan Sprague as a POSIX compliant C wrapper for the *iRobot Open Interface*. Together with Stefan Sander, Michal Policht, and Brandon Fosdick's *QextSerialPort*[23] we created a C++ version of COIL which compiles on Windows, Linux, and OS X. COIL opens a serial port and directly communicates with the *iRobot Open Interface*. All the functions of the interface have been ported in our version, with some additional useful functions.

```
// Allocate and start COIL
coil = new COIL("COM4");

coil->startOI();
coil->enterFullMode();

// Drive forward one meter, checking for collision at intervals
coil->wheelDrive(100,100); // 100 mm/s
coil->waitTime(10000); // 10 seconds
for(int i = 0; i < 10; i++) {
     if(coil->getBumpsAndWheelDrops() != 0) break;
     coil->wheelDrive(100, 100); // 100 mm/s
     coil->waitTime(100); // 0.1 seconds
}
coil->wheelDrive(0,0); // stop moving

// Turn around
coil->turn(180);

// Shutdown and cleanup
coil->stopOI();
delete COIL;
```

Code 5-6: Simple COIL Usage Example – Moving the Robot

[22] For more information about COIL, please visit http://code.google.com/p/libcreateoi.
[23] For more information about QextSerialPort, please visit http://qextserialport.sourceforge.net.

5.3.3 Controller

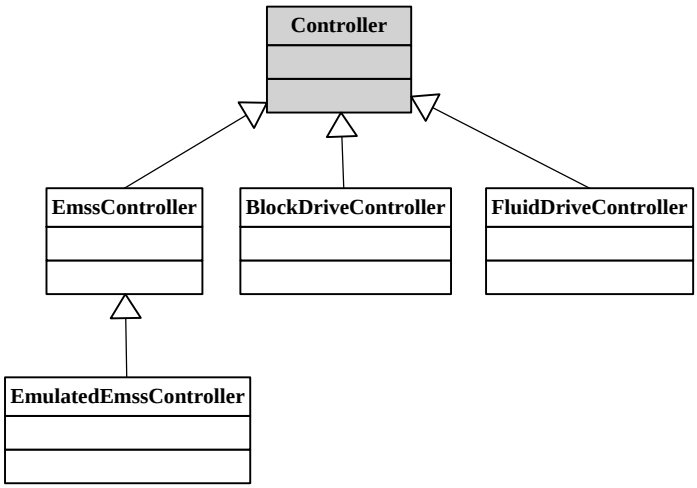


Figure 5-5: Controller Model

The abstract Controller class is the base class of fully functional Controllers. It provides the virtual methods *run()* and *emergencyStop()* which must be implemented by the child classes, and inherits from QThread, a cross-platform execution fork. The process routine of a Core Controller is the *run()* method and must only exit if the variable *stopRequested* is *true*. Currently, there are three different Controller implementations: Block Drive Controller, Fluid Drive Controller, and Emss Controller. Only the later is meant for real use, the others are for testing and diagnostic purposes only. In addition to sending movement commands to COIL, the Controller is also responsible for passing along sensor data to other components such as the Movement Tracker. This is realized in the form of *Qt* signals.

5.3.3.1 Block Drive Controller

The exact purpose of the Block Drive Controller is encoded in its name. It provides a function for moving forward, moving backward, turning right, and turning left with a predefined speed, angle, and distance. The *run()* method of the Controller thread executes the set action and sends the corresponding commands to COIL. The Block Drive Controller also offers an option for a slow start. Slow start can be set to linear or square. This means that the speed increases to its predefined value in linear or square fashion. This is attached for more exactness of driven distance. The disadvantage of this Controller is its blocking nature. This means that calling *move(speed,distance)* with speed set to 100 mm/s and distance set to 1000 mm will make the calling thread block for one second.

5.3.3.2 Fluid Drive Controller

The Fluid Drive Controller is a further development of the Block Drive Controller. It supports control of the robot with a two-dimensional GUI joystick. The Controller has two yokes, *yokeX* and *yokeY* which accept values between -1 and 0. The speed of the robot is determined by the *yokeY* value, while the direction of the robot's movement is set by the *yokeX* value. Precisely put, the wheel speeds are given by

$$wheel_L = yoke_y \cdot speed_{\max} - yoke_x \cdot speed_{\max}$$

$$wheel_R = yoke_y \cdot speed_{\max} + yoke_x \cdot speed_{\max}$$

The disadvantage of this controller is that rotations around the center axis of the robot are not supported due to the nature of the joystick. For a visual representation of how the Fluid Drive Controller works, please see Figure 5-3.

5.3.3.3 Emss Controller

This Controller is the most important Controller in our project. Its main function is to receive the differential steering commands and to pass along sensor data to other Core components. The following code snippets originate from the *run()* method of the Emss Controller.

```
// Get movement...
double distanceDelta = coil_getDistance();
double angleDelta = coil_getAngle();

// Emit signals for movement tracker
emit signalMovedDistance(distanceDelta);
emit signalChangedAngle(angleDelta);
```

Code 5-7: Forwarding Change of Distance and Angle Data

In this code the driven distance and the turned angle are prompted and the result sent as a signal. These signals are intercepted by the active Movement Tracker.

```
// Get other sensor data
int sharpIRSensor = coil_getAnalogSensorDistance();
int bumpsWheelDrop = coil_getBumpsAndWheelDrops();

// Collision?
if (((COIL::BUMPWHEELDROP_BUMP_LEFT & bumpsWheelDrop) ==
       COIL::BUMPWHEELDROP_BUMP_LEFT) ||
       ((COIL::BUMPWHEELDROP_BUMP_RIGHT & bumpsWheelDrop) ==
              COIL::BUMPWHEELDROP_BUMP_RIGHT)) {
       emit signalCollision();
       if(core->boolSetting("EMSSCONTROLLER_EMERGENCY_STOP_ENABLED")
              == true) emergencyStop();
}

// Object detected?
if (sharpIRSensor <
core->intSetting("EMSSCONTROLLER_SHARP_IR_SENSOR_CUTOFF_VALUE")) {
       emit signalObjectDetected(sharpIRSensor, 0); // Angle is 0
                                    because it is straight ahead always!
       if(core->boolSetting("EMSSCONTROLLER_EMERGENCY_STOP_ENABLED")
              == true && sharpIRSensor <
core->intSetting
       ("EMSSCONTROLLER_SHARP_IR_SENSOR_EMERGENCYSTOP_BUFFER_MM"))
       emergencyStop();
}
```

Code 5-8: Controller Collision and Obstacle Detection

This is the code to check whether there has been a collision or if an obstacle is detected. The Maps, such as the Heat Map, receive a collision or object signal and register them in their data structures. Depending on the settings, the robot performs an emergency stop if a collision or obstacle is detected.

```
// Send wheel speeds to COIL
```

```
            if(mode != EmssController::EmergencyStop)
                  coil_directDrive(Lwheel, Rwheel);

            // Sleep our interval...
            this->msleep(interval);

      }
```

Code 5-9: Controller Movement Commands Sent to COIL

These commands pass the steering information of each wheel to COIL and subsequently wait for the next process execution. Currently, certain features from both the Block Drive Controller and Fluid Drive Controller are implemented within the Emss Controller, creating a set of Emss Controller modes such as *Idle*, *Move*, *Turn*, *Joystick*, *WheelDrive*, *EmergencyStop*. However, the only mode not used for diagnostic purposes (i.e. manually driving out of a corner) is the *WheelDrive* mode. For a visual representation of how the Emss Controller works, please see Figure 5-4Figure 5-3.

5.3.3.4 Emulated Emss Controller

The Emulated Emss Controller has exactly the same functionality as the Emss Controller. The only difference is that COIL commands are not actually forwarded to COIL. These commands are emulated internally within the Controller. This Controller was developed to run the *emss* Core without a physical robot but still be able to perform navigational tests, algorithms, et cetera.

To maximize code reuse and implementation efficiency, the Emulated Emss Controller extends from the Emss Controller, overwriting only the methods needed for emulation. This was achieved by wrapping the COIL commands *directDrive(left,right)*, *getDistance()*, *getAngle()*, *getAnalogSensorDistance()*, and *getBumpsAndWheelDrops()* in the Emss Controller. Thus, when the Emss Controller wants to call *getDistance()* on COIL, instead it calls the local method *coil_getDistance()*. In the Emss Controller this function is defined simply as:

```
double EmssController::coil_getDistance() {
     return (double) core->coil->getDistance();
}
```

Code 5-10: Emss Controller Change of Distance Wrapper

However, the Emulated Emss Controller overwrites this method, emulating the behavior of the *iRobot Create Interface*.

```
double EmulatedEmssController::coil_getDistance() {
            // Get the elapsed time since last call
            double elapsedTimeSinceGetDistance =
                  (double)lastGetDistanceTimestamp.restart();

            // We need to return a value (mm) based on the wheel speed (mm/s) and
               elapsed time (ms)...
            double wheelSpeed = ((double)emulatedLwheel +
                  (double)emulatedRwheel) / 2.0;  // Average it out
            double distance = (wheelSpeed *
                  (elapsedTimeSinceGetDistance / 1000.0));
            return distance;
}
```

Code 5-11: Emulated Emss Controller Change of Distance Emulation

This above code emulates the moved distance of the robot. First it measures the time between now and the last call of this method. The average between the left wheel speed and the right wheel speed is calculated and the result is the multiplication of the average and elapsed time.

Emulating the changed angle is not as trivial, as the *iRobot Create Interface* does not disclose how this is done. We have implemented two different methods for emulated the changed angle. The first is based on the linear motion of each wheel, and becomes increasingly inaccurate as the elapsed time increases. The later is based on the theory discussed in Section 4.6.

```
double EmulatedEmssController::coil_getAngle() {

	// Get the elapsed time since last call
	double elapsedTimeSinceGetAngle = (double)lastGetAngleTimestamp.restart();

	if(calculateAngleBasedOnRotationalSpeed == true) {

		// Calculate the change of angle based on the rotational wheel speed
		// ((distance / 2 PI r) * 360)

		double angle = 0.0;
		double radius = core->doubleSetting("ROBOT_WHEEL_OFFSET_MM");
		double distanceL = emulatedLwheel *
			(elapsedTimeSinceGetAngle / 1000.0);
		double distanceR = emulatedRwheel *
			(elapsedTimeSinceGetAngle / 1000.0);
		double circumference = 2*PI*radius;
		double angleL = (distanceL / circumference) * 360.0;
		double angleR = (distanceR / circumference) * 360.0;
		angle = angleR - angleL;
		angle *= PI/2;
		return angle;

	} else {

		// Calculate change of angle using physics

		if (emulatedLwheel != emulatedRwheel) {
			double arclenL = emulatedLwheel *
				(elapsedTimeSinceGetAngle / 1000.0);
			double d = core->doubleSetting("ROBOT_WHEEL_OFFSET_MM") * 2;
			double radiusL = (emulatedLwheel*d) /
				(emulatedRwheel-emulatedLwheel);
			double angleAlpha = Deg(arclenL / radiusL);
			double angleBeta = 180.0 - 90.0 - angleAlpha;
			double angleGamma = 90.0 - angleBeta;
			return angleGamma;
		}
		return 0.0;

	}
}
```

Code 5-12: Emulated Emss Controller Change of Angle Emulation

5.3.4 Movement Tracker

Responsible for tracking movements of the robot, and in turn performing the localization, a Movement Tracker accepts signals from the active Controller and translates them accordingly. Furthermore, Trackers also forward changes about the robot position to other components, especially the Maps, in forms of signals. The Movement Tracker is an abstract class which needs to be defined by subclasses. The only significant variable is *transformation*, which is a Trafo2D object. The input slots, i.e. signals received from the Controller, are *registerMovedDistance(distance)*, *registerChangedAngle(angle)*, *registerCollision()*, and *registerObjectDetected(distance,angle)*.

The current *emss* software offers only one implementation of Movement Tracker, namely the Raw Movement Tracker. However, in the future other, more advanced, Movement Trackers will be implemented in order to increase the localization precision.

5.3.4.1 Raw Movement Tracker

The Raw Movement Tracker tracks the robot's movement by geometrically interpreting the sensor data sent back from the robot. This method remains accurate (at least as accurate as the sensor data) as long as the robot's movement is linear, meaning that the wheels were not differentially driven. When a differential steering system is made use of, the accuracy of the tracker declines as the sensor interval increases. However, we have found that this implementation of the tracker is surprisingly sufficient, as shown in Figure 5-6.

```
void RawMovementTracker::registerChangedAngle(double angle){
     if(angle != 0) {
            transformation = transformation * Trafo2D::rot(Rad(angle));
            emit moved(this->x(), this->y(), this->rotation());
     }
}
```

Code 5-13: Raw Movment Tracker Change of Angle Processing

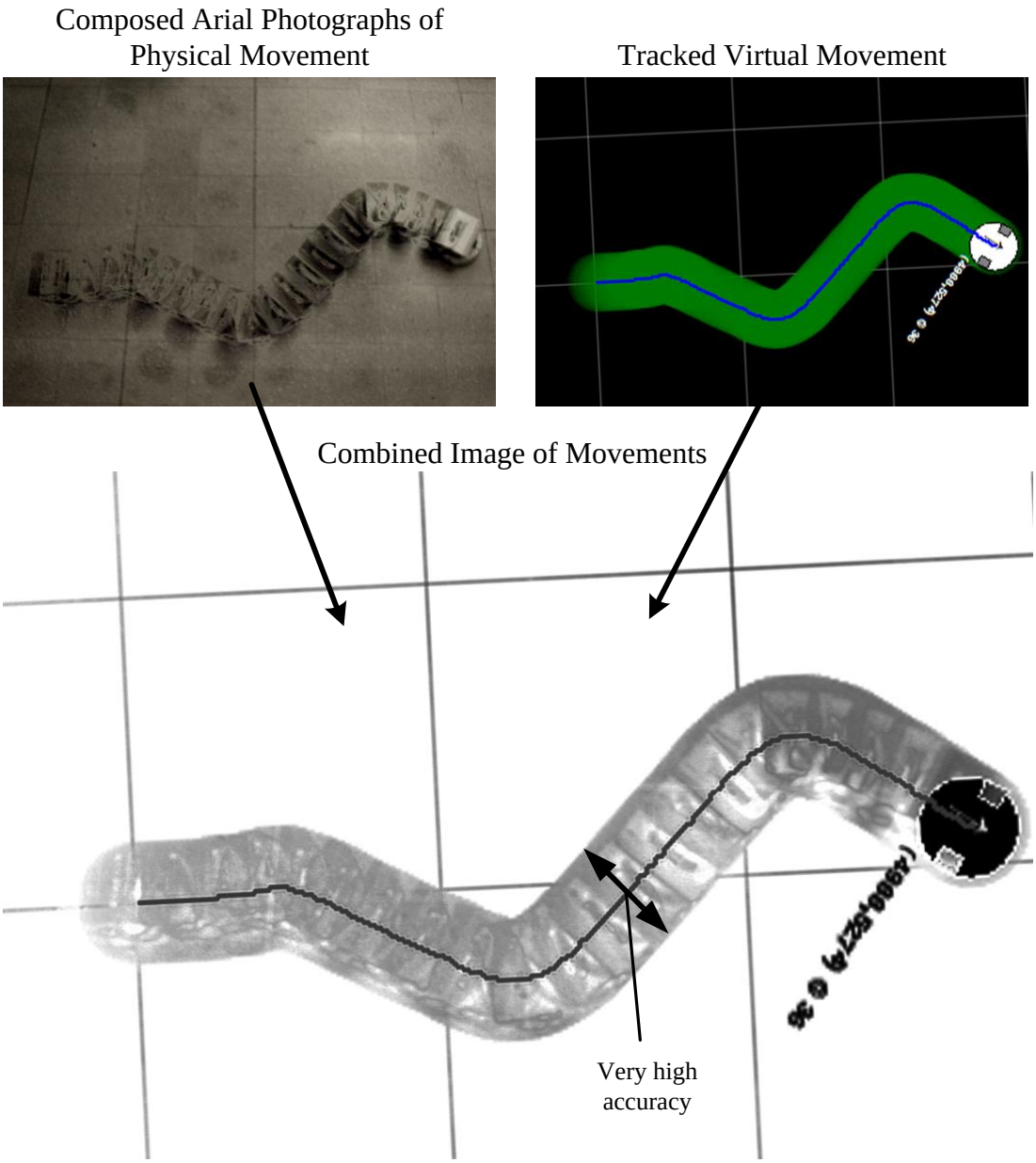


Figure 5-6: Raw Movement Tracker Accuracy Test Composition

5.3.5 Navigation

The Navigation class holds the information of the navigation points, or way points, for the robot. Other data structures, such as splines, are also included in this class. Tasks who wish to navigate from point to point or via spline use the Navigation component as an abstraction to do so. The most important methods are *calculateLeftRightWheelSpline()* and *getWheelSpeed(tick,interval)* which have been discussed in detail in Section 4.5.1.

```
void Navigation::calculateLeftRightWheelSpline() {

    double wheelNodeDistanceBetweenNode = targetSpeed *
           ((double) interval / 1000.0);

    // Init and drop nodes
```

```
    wheelLeftSplineX->clearNodes();
    wheelLeftSplineY->clearNodes();
    wheelRightSplineX->clearNodes();
    wheelRightSplineY->clearNodes();

    // Calculate new nodes for wheel splines
    for (int navNode = 0; navNode < navSplineX->getNodeCount() - 1; navNode++) {

        // Get the distance between the two nav nodes
        double distanceBetweenNodes =
            getDistanceBetweenNavNodes(navNode, navNode+1);

        // Create the wheel nodes at equal distance between the two nodes
        for (int wheelNode = 0; wheelNode <
          (distanceBetweenNodes / wheelNodeDistanceBetweenNode); wheelNode++) {
            // Get the t value for the nav node, which is discretisized
               based on the wheel node distance between nodes
            double t = (double) wheelNode / (double)
                (distanceBetweenNodes / wheelNodeDistanceBetweenNode);

            // Get the node positions for both the left and right wheel
            // splines
            Vector2D c(navSplineX->getValue(navNode, t),
                navSplineY->getValue(navNode, t));
            Vector2D c1(navSplineX->getFirstDerivative(navNode, t),
                navSplineY->getFirstDerivative(navNode, t));
            Vector2D v = c1 / norm(c1);
            Vector2D n = Vector2D(-v.y(), v.x());
            Vector2D l = c + (wheelOffset * n);
            Vector2D r = c - (wheelOffset * n);

            wheelLeftSplineX->addNode(l.x());
            wheelLeftSplineY->addNode(l.y());
            wheelRightSplineX->addNode(r.x());
            wheelRightSplineY->addNode(r.y());
        }
    }
}
```

Code 5-14: Navigation Wheel Spline Calculation

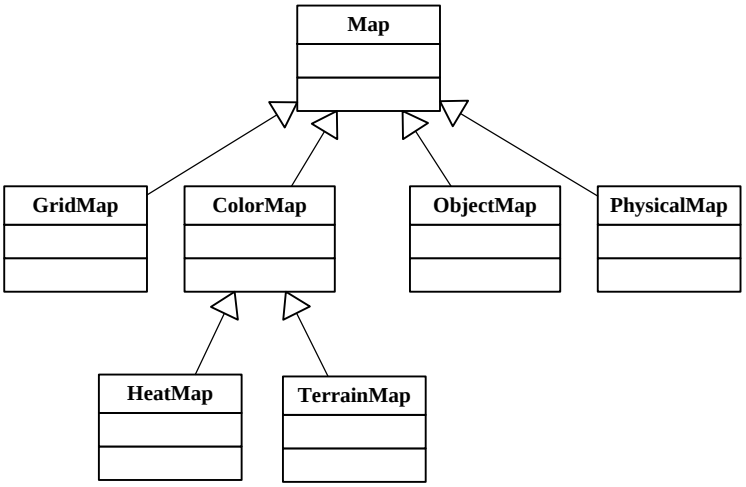
The *getWheelSpeed(tick,interval)* method requires two parameters: *tick* and *interval*. The *tick* parameter is straightforward and is used to calculate which node in the wheel splines will be used to extract the current wheel speed. The *interval* parameter is used to insure that the requesting Task has the same interval as used for creating the wheel splines. However, if the setting *NAVIGATION_DYNAMIC_INTERVAL_ENABLED* is *true*, *getWheelSpeed(tick,interval)* checks to see what the actual interval was between the Tasks request for wheel speeds, and, if this interval has drifted too far from the original interval used to create the wheel splines, they are accordingly recalculated. The actual calculation of the wheel speeds are only the last four lines of the *getWheelSpeed(tick,interval)* method.

```
Vector2D Navigation::getWheelSpeed(int tick, int interval) {

    bool rebuildWheelSplines = false;
```

```
    int actualInterval = interval;

    // Get the node based on the tick, and the current offset, so that you can
    // start navigation in the middle or so
    int node = tickOffset + tick;
    double t = 0.0;

    //Are we done?
    if (node >= wheelLeftSplineX->getNodeCount() - 1) {
          return SplineNavigationWheelSpeedFinished;
    }

    // Dynamic or fixed interval?
    if(dynamicIntervalEnabled == true) {

          // Dynamic interval - adjust spline for incomming interval but only
          // rebuild if exceeds flexibility
          actualInterval = lastGetWheelSpeedTimestamp.restart();

          // Rebuild spline?
          if(actualInterval > interval + intervalFlexibility ||
                actualInterval < interval - intervalFlexibility) {

                //Debug::print("[Navigation] Interval flexiblity exceeded by
                // %1", interval - actualInterval);
                rebuildWheelSplines = true;
          }

    } else {

          // Fixed interval, nothing to do except warning and return if
          // different
          if (this->interval != interval) {
                return SplineNavigationWheelSpeedFinished;
          }
    }

    // Rebuild wheel splines because of interval delay or node offset?
    if(rebuildWheelSplines == true) {
          interval = actualInterval;
          calculateLeftRightWheelSpline();
    }

    // Calculate speed in mm/s per wheel
    double vl = norm(Vector2D(wheelLeftSplineX->getFirstDerivative(node, t),
          wheelLeftSplineY->getFirstDerivative(node, t)));
    double vr = norm(Vector2D(wheelRightSplineX->getFirstDerivative(node, t),
          wheelRightSplineY->getFirstDerivative(node, t)));
    vl *= 1.0 / (interval / 1000.0);
    vr *= 1.0 / (interval / 1000.0);
    return Vector2D(vl, vr);
}
```

Code 5-15: Navigation Wheel Speed Calculation

5.3.6 Map



Maps hold information about the environment. In the *emss* framework there exist several types of maps, each serving a different purpose. *emss* Maps must define the abstract class Map, however they enjoy a significant amount of freedom. The only virtual Map method is *paint(view,scale)*, which is used by the Viewports for displaying the map to the user. In addition, any map must provide a *width* and a *height*. If a Map does not know its own dimensions, it can consult the emss Core for the basic world boundaries. A certain set of functionality shared by both the Heat Map and Terrain Map have been combined into the Color Map. Self-explanatory Maps, such as the Grid Map and Object Map, will not be discussed in further details.

As discussed in the introduction of Chapter 0, Maps is an example where the data tier and presentation tier have been merged into a single concept. We chose to do this as the presentation, i.e. the visual image of the Map, is typically identical to the data. Of course, the Map *paint(view,scale)* method could be refactored to a presentation "wrapper" class in a separate package, however we see no benefit of such a bloated design.

5.3.6.1 Heat Map

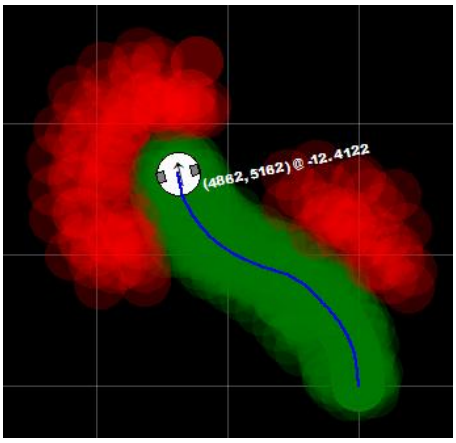


Figure 5-7: Heat Map Example

A Heat Map is built to visualize the discovered environment of the robot. When the robot moves through its environment, the Movement Tracker is responsible for translating the sensory data into

physical movements through space. In turn, the Movement Tracker sends signals to the appropriate Maps, such as the Heat Map, registering various kinds of "heat". The *emss* Heat Map support two kinds of heat: *Open Area* and *Collision Area*. *Open Area* is displayed as green, and portrays the robots path through the environment, while *Collision Area* is displayed as red, and represents any sort of collision or obstacle, forming the environments boundary. The Heat Map can be consulted by different *emss* components in order to make decisions, such as navigational choices. In addition, other influences, such as a user, can also "color" the Heat Map for diagnostic reasons such as staking out a territory.

```
// Robot collided at x,y
heatMap->registerHeat(ColorMap::CollisionAreaChannel, x, y, heatSpotOpacity,
     heatSpotSize);
```

Code 5-16: Example for Registering a Collision on the Heat Map

```
// Look ahead if it is good to move using a sweep across a start and end angle...
double heat = 0.0;
int sweepAngle = 30;
double lookAheadRange = 300; // 30cm
for(int angle = -sweepAngle/2; angle <= sweepAngle/2; angle++) {
  Trafo2D checkpoint = core->movementTracker->transformation *
     Trafo2D::rot(-Rad(angle)) * Trafo2D::trans(0, lookAheadRange);
  heat += core->heatMap->getChannelValue(HeatMap::CollisionAreaChannel,
     checkpoint.trans().x(), checkpoint.trans().y());
}

if(heat == 0.0) {
  // Safe to move...
}
```

Code 5-17: Example Code for Consulting a Heat Map

5.3.6.2 Terrain Map

Currently, the Terrain Map is only used for displaying the results of the algorithm described in Section 4.7. However, the data structure is now ready to be used by future algorithms. The class provides three main functions: *raise(x,y)*, *lower(x,y)*, and *getHeight()*. The higher an area of the map is, the brighter white it is colored. The *raise(x,y)* and *lower(x,y)* methods are overloaded to allow entire sections of a map to be altered at once.

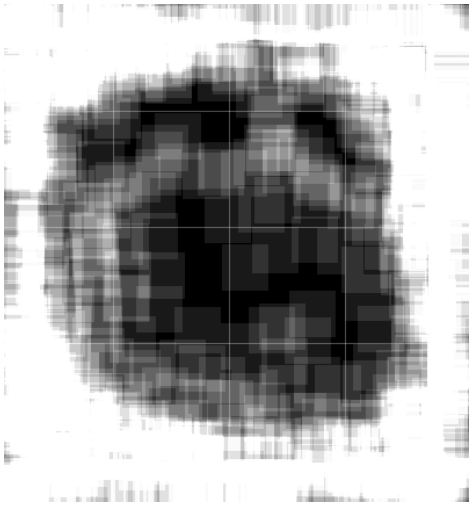
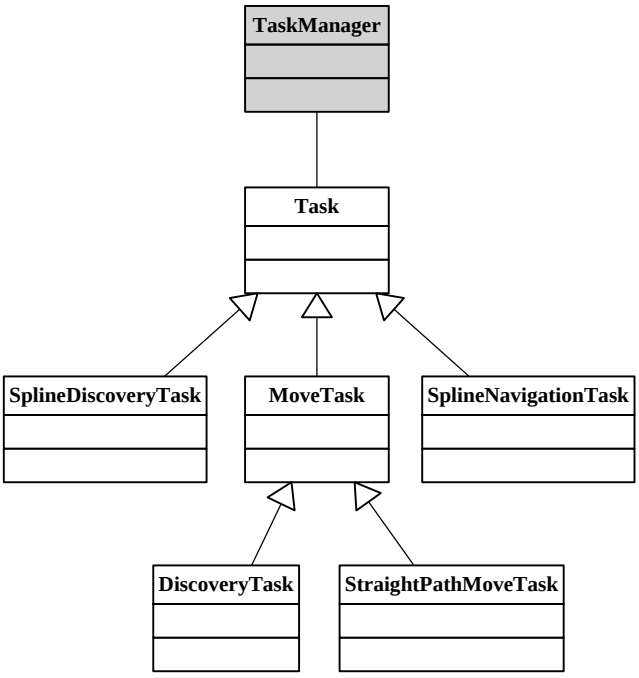


Figure 5-8: Example Height Map Surrounding the Robot

5.3.6.3 Physical Map

The *emss* Physical map represents the real world environment in its physical state. This Map is currently used for simulation purposes, but in the future can also be used for various Tasks which would like to combine a predefined perception of the environment. The Physical Map offers the method *getAreaType(x,y)* which returns either *Open Area* or *Collision Area* – just like the Heat Map. The Emulated Emss Controller uses this Map to emulate the IR Range Finder during simulations, something which has proven especially useful as the robots environment can now be almost entirely emulated. To define the areas of a physical map, all one has to do is draw a RGBA image, such as a 24-bit PNG, and let the Physical Map load the file. Any area in the image which has the alpha channel not at 255 (full) is regarded as *Open Area*.

5.3.7 Tasks



Core Tasks bind all the components of the *emss* framework together, enabling the robot to perform actions in its environment. As already mentioned, the Task Manager is responsible for executing the Tasks one by one. Each *emss* Task must inherit from the class Task and implement the method *process()*. When a Task is created, its initial status is *Waiting*. When the Task Manager decides to schedule the Task for execution, its status is set to *Running*. The *process()* method of the Task is called by the Task Manager at its given *interval* (each Task can define its own *interval*) until the task sets its status as *Finished*.

5.3.7.1 Task Manager

The Task Manager is a separate thread which contains a Task List. The Task List is nothing more than a helper class for sequentially storing and retrieving Tasks, offering useful functions such as *getNextSpecificTask(type)*. As soon as the Task Manager receives a Task, appends it to the Task List and starts to execute it. If the Task Manager receives more than one Task at a time, it appends it to the back of the Task List, where in time will eventually be executed. The Task Manager also allows Tasks to be interrupted. In this case the manager will set the current Task as *Interrupted* and go to the next Task in the list.

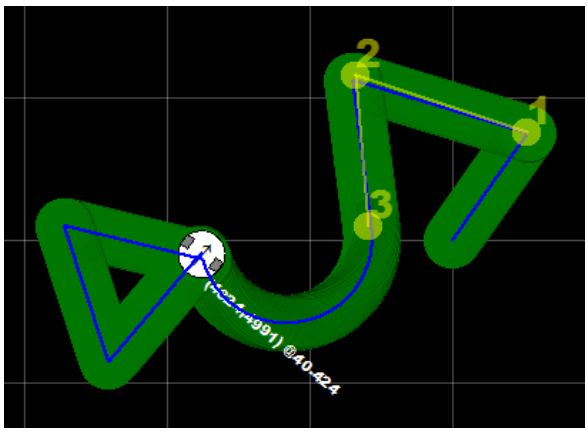


Figure 5-9: Execution of a Series of Tasks

5.3.7.2 Test Move Task

Several test movements are provided by the Test Move Task. These test movements are used for diagnostic purposes, such as calibration, accuracy observation, and general research. The following self-explaining test moves are available: *Square, Rotate 90, Rotate 360, Triangle, Straight, Circle,* and *Vector Circle*.

5.3.7.3 Straight Path Move Task

This Task moves the robot in a linear fashion to the given destination. Multiple Straight Path Move Tasks can be appended to the Task Manager in order to create a series of chunky movements.

5.3.7.4 Spline Navigation Task

Spline navigation is very trivial. All it does is drive through all the navigational points contained in the Navigation component in the form of a spline by calling the Navigation methods *getWheelSpeed(tick,interval)*. There is some additional functionality added, such as aligning the robot with the spline before navigation, and resetting the navigation points when finished.

```
void SplineNavigationTask::process() {

     // Align the robot and then crawl the spline...
     if(tick == 0 &&
       core->boolSetting("SPLINENAVIGATIONTASK_ALIGN_ROBOT_WITH_SPLINE_") == true)
            ((EmssController*) core->controller)->turn(
                  (int)(- core->movementTracker->rotation() +
                  core->navigation->getAngleForSplineAlignment()),
                  core->intSetting("
                        SPLINENAVIGATIONTASK_ALIGN_ROBOT_WITH_SPLINE_SPEED"));

     // Send wheel speeds for current tick to controller
     Vector2D wheelSpeed = core->navigation->getWheelSpeed(tick, interval);
     ((EmssController*) core->controller)->wheelDrive((short)wheelSpeed.x(),
            (short)wheelSpeed.y());

     // Finished?
     if(wheelSpeed == SplineNavigationWheelSpeedFinished) {
            // Reset?
            if(core->boolSetting("
                  SPLINENAVIGATIONTASK_RESET_NAV_POINTS_WHEN_DONE") == true)
                  core->navigation->clearNavPoints();
            else
                  core->navigation->setTickOffset(tick);

            status = Task::Finished;
     }

     tick++;
}
```

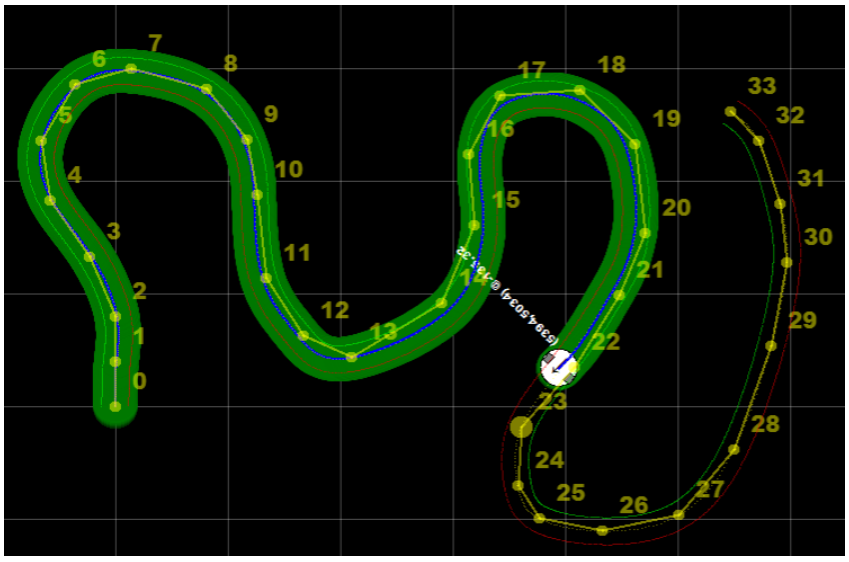


Figure 5-10: Execution of Spline Navigation Task with Multiple Navigation Points

5.3.7.5 Discovery Task

This is a simple implementation of the classic robot discovery algorithm. The robot moves straight until it reaches a collision. Upon reaching a collision it rotates until it is free to move. This algorithm

works well for certain environments for finding the bounds (i.e. walls), but quickly finds itself stuck in a loop of a specific path.

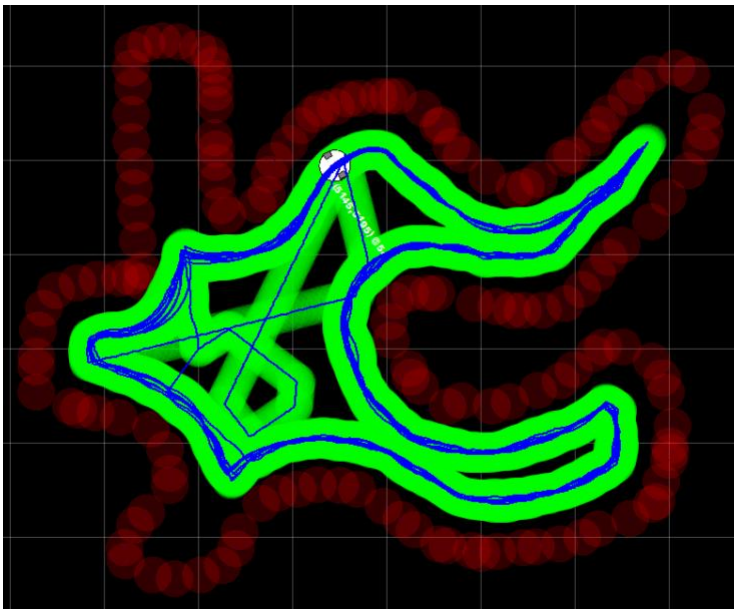


Figure 5-11: Execution of Discovery Task

5.3.7.6 Spline Discovery Task

The Spline Discovery Task is the implementation of the theory discussed in Section 4.7. For navigating in an unknown environment a height map is used as mentioned above. This is a rather complex procedure, and is presented along with the code. For further details or clarifications on the algorithm please see Section 4.7.

This first section is important. First we check if there are any nodes in the spline and if the environment is not already explored. If it is already explored the algorithm is not required and will be canceled. Otherwise new terrain cut points are added:

```
void SplineDiscoveryTask::process() {

    // Are there nodes in the spline?
    if (node >= core->navigation->wheelLeftSplineX->getNodeCount() - 1) {

        if (isExplored() == true) {
            ((EmssController*) core->controller)->wheelDrive(0, 0);
            Task::status = Task::Finished;
            return;
        } else {
            core->navigation->clearNavPoints();
            addNewTerrainCutPoints();
            node = 0;
            return;
        }

    }
```

Code 5-18: Spline Discovery Process Loop

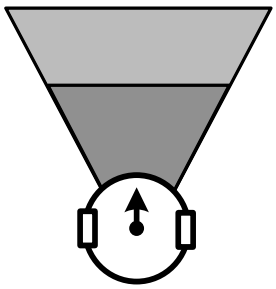


Figure 5-12: Spline Discovery Areas of Collision

This next part of the code checks the heat map whether it is safe to move ahead. This is done by directly checking ahead of the robot from a given start and end angle. Figure 5-12 presents two areas of collision. If an obstacle in the dark area is detected the robot will immediately stop and turn ninety degrees. This is to avoid a head-on collision with the obstacle. If an obstacle in the light area is detected, new terrain cut points are calculated. This discrimination lets the calculation of new terrain cut points result in a natural collision avoidance per algorithm.

```
// Look ahead if it is good to move there using a sweep across a start and
// end angle...
double heat = 0.0;
for (int i = 0; i < core->longSetting("SPLINEDISCOVERYTASK_RANGE_MM");
        i += 10) {
       for (int angle = -(core->longSetting("
              SPLINEDISCOVERYTASK_SWEEP_ANGLE") / 2);
              angle <= (core->longSetting("
              SPLINEDISCOVERYTASK_SWEEP_ANGLE") / 2);
              angle++) {
              Trafo2D checkpoint = core->movementTracker->transformation *
                     Trafo2D::rot(-Rad(angle)) * Trafo2D::trans(0, i);
              heat += core->heatMap->getChannelValue(
                     HeatMap::CollisionAreaChannel,
                     (long) ((checkpoint.trans().x()) / core->scale),
                     (long) ((checkpoint.trans().y()) / core->scale));
              if (i < core->longSetting("
                SPLINENAVIGATIONTASK_CRITICAL_AREA_DISTANCE_MM") && heat > 0)
              {
                     collisionInCriticalArea = true;
              }
       }
}
// Safe to move?
if (heat > 0.0 && lastProcessWasCollision == false) {
       // is the collision in the critical area?
       if (collisionInCriticalArea == false) {
              core->navigation->clearNavPoints();
              addNewTerrainCutPoints();
              node = 0;
              lastProcessWasCollision = true;
       } else {
              // turn away from the collision
              ((EmssController*) core->controller)->turn(90,
                     core->doubleSetting("NAVIGATION_TARGET_SPEED_MMPS"));
              core->navigation->clearNavPoints();
              addNewTerrainCutPoints();
```

```
                node = 0;
                lastProcessWasCollision = true;
                collisionInCriticalArea = false;
        }
    } else {
        lastProcessWasCollision = false;
    }
```

Code 5-19: Spline Discovery Collision Avoidance

With the following code the steering information is sent to the controller:

```
    // Align the robot and then crawl the spline...
    Vector2D wheelSpeed = core->navigation->getWheelSpeed(node, interval);
    ((EmssController*) core->controller)->wheelDrive((short) wheelSpeed.x(),
        (short) wheelSpeed.y());
    node++;
}
```

Code 5-20: Extracting Steering Information from Spline

The most interesting part in this class is how the terrain cut points are calculated, as discussed in the Chapter "Theory". First the height array is filled with the value for an unknown height:

```
    int heights[mapWidth][mapHeight];
    int explorationMap[mapWidth][mapHeight];

    long robotX = core->movementTracker->x();
    long robotY = core->movementTracker->y();

    // Fill array with default values
    for (int x = 0; x < mapWidth; x++) {
        for (int y = 0; y < mapHeight; y++) {
            heights[x][y] = TERRAIN_UNKNOWN_HEIGHT;
        }
    }
```

Code 5-21: Initialization of the Local Height and Exploration Map

Then the exploration map is filled with the information from the Heat Map. That means that every point is assigned to either *discovered*, *obstacle* or *unknown*:

```
    // Fill the exploration array
    for (int x = 0; x < mapWidth; x++) {
        for (int y = 0; y < mapHeight; y++) {
            long heatMapX = robotX - RELATIVE_EXPLORATION_WIDTH / 2 +
                x * EXPLORATION_MAP_GRID_SIZE;
            long heatMapY = robotY - RELATIVE_EXPLORATION_HEIGHT / 2 +
                y * EXPLORATION_MAP_GRID_SIZE;
            if (core->heatMap->getChannelValue(HeatMap::OpenAreaChannel,
               heatMapX / core->scale, heatMapY / core->scale) > 0.0) {
                explorationMap[x][y] = Discovered;
            } else if (core->heatMap->getChannelValue(
               HeatMap::CollisionAreaChannel, heatMapX / core->scale,
               heatMapY / core->scale) > 0.0) {
```

```
                        explorationMap[x][y] = Obstacle;
                } else {
                        explorationMap[x][y] = Unknown;
                }
        }
}
```

Code 5-22: Determining the Exploration Status

The algorithm begins with the robots position and calculates for each point the heights depending on the exploration map. As a result, points near an obstacle, or who are marked as discovered, get a low height, and unknown points get higher and higher in height, the further away they are from a known point.

```
int h;
QQueue<TerrainPointT<int> > queue;
queue.enqueue(TerrainPointT<int> (mapWidth / 2, mapHeight / 2, 0));

// Calculate the heights
while (!queue.isEmpty()) {
        TerrainPointT<int> p = queue.dequeue();

        if (p.height + 1 > TERRAIN_MAP_MAX_HEIGHT) {
                h = TERRAIN_MAP_MAX_HEIGHT;
        } else {
                h = p.height + 1;
        }

        for (int k = -1; k <= 1; k++) {
                for (int j = -1; j <= 1; j++) {
                        if (p.x + k < mapWidth && p.x + k >= 0 &&
                                p.y + j < mapHeight && p.y + j >= 0 &&
                                (heights[p.x + k][p.y + j] > h ||
                                heights[p.x + k][p.y + j] ==
                                TERRAIN_UNKNOWN_HEIGHT)) {
                                switch (explorationMap[p.x + k][p.y + j]) {
                                case Obstacle:
                                        heights[p.x + k][p.y + j] = 0;
                                        queue.enqueue(TerrainPointT<int> (p.x + k,
                                                p.y + j, 0));
                                        break;
                                case Discovered:
                                        heights[p.x + k][p.y + j] = 0;
                                        queue.enqueue(TerrainPointT<int> (p.x + k,
                                                p.y + j, 0));
                                        break;
                                case Unknown:
                                        heights[p.x + k][p.y + j] = h;
                                        queue.enqueue(TerrainPointT<int> (p.x + k,
                                                p.y + j, h));
                                        break;
                                }
                        }
                }
```

```
        }
    }
```

Code 5-23: Determining the Heights

All points which correlate to the given cut level point are found and added to a queue:

```
std::vector<QPoint> terrainCutPoints;

// Add the points with the terrain cut height to the spline
while (terrainCutPoints.size() < 3) {
       for (int x = 0; x < mapWidth; x++) {
              for (int y = 0; y < mapHeight; y++) {
                     if (heights[x][y] == TERRAIN_CUT_LEVEL) {
                            long navPointX = robotX -
                            RELATIVE_EXPLORATION_WIDTH / 2 +
                            x * EXPLORATION_MAP_GRID_SIZE;
                            long navPointY = robotY -
                            RELATIVE_EXPLORATION_HEIGHT / 2 +
                            y * EXPLORATION_MAP_GRID_SIZE;
                            terrainCutPoints.push_back(QPoint(navPointX,
                                   navPointY));
                     }
              }
       }
       cutLevel--;
}
```

Code 5-24: Intersecting the Results Using a Cut Level

The cut level points are in an unsorted list. To make use of these points it is necessary to sort the list and remove redundant points. For this reason a minimal distance between the cut points is defined. All points which do not have this distance to the last point will removed from the list:

```
// Sort the terrain cut points
for (int i = 0; i < terrainCutPoints.size(); i++) {

       int distance = INT_MAX;
       int nearestPoint = 0;
       for (int k = i + 1; k < terrainCutPoints.size(); k++) {
              int temp = std::abs(terrainCutPoints[i].x() -
                     terrainCutPoints[k].x()) +
                     std::abs(terrainCutPoints[i].y() -
                     terrainCutPoints[k].y());
              if (temp < distance) {
                     distance = temp;
                     nearestPoint = k;
              }
       }
       if (i != terrainCutPoints.size() - 1) {
              QPoint temp = terrainCutPoints[i + 1];
              terrainCutPoints[i + 1] = terrainCutPoints[nearestPoint];
              terrainCutPoints[nearestPoint] = temp;
       }
```

```
}
```

Code 5-25: Sorting the Result Set

Finally, the calculated points are registered in the Navigation component.

```
// Add the terrain cut point to the navigation spline
QPoint lastPointAdded;
for (int i = 0; i < terrainCutPoints.size(); i++) {
	// Take only the cut points with the distance CUT_POINT_DISTANCE from
	// the last point
	if (i == 0) {
		this->core->navigation->addNavPoint(terrainCutPoints[i].x(),
			terrainCutPoints[i].y());
		lastPointAdded = terrainCutPoints[i];
	}
	if (i != 0 && i != terrainCutPoints.size() - 1 &&
		std::sqrt(std::pow(lastPointAdded.x() -
		terrainCutPoints[i].x(), 2.0) +
		std::pow(lastPointAdded.y() - terrainCutPoints[i].y(),
		 2.0)) >core->intSetting("
		SPLINEDISCOVERYTASK_TERRAIN_CUT_POINT_DISTANCE_MM")) {
		this->core->navigation->addNavPoint(terrainCutPoints[i].x(),
			terrainCutPoints[i].y());
		lastPointAdded = terrainCutPoints[i];
	}

}
```

Code 5-26: Creating a Navigation Path for Traversal

5.3.8 GUI

The GUI packet consists of different widget classes which can be used in an *emss* application to control the robot, display information of the Core, set properties, et cetera. Most of these classes are quite generic and can be re-used in other projects.

5.3.8.1 Viewport

The Viewport class is a very important part of the *emss* user interface. A Viewport has the ability to display any *emss* Map as well as the Map Objects. In addition to displaying Maps, a Viewport can accept actions from the user and provides the necessary tools for navigating the Maps, such as scrolling. A Viewport allows multiple Maps to be layered on-top of each other, and automatically generates the proper GUI elements for hiding and showing specific Maps. Core Maps can have multiple viewports attached to it, allowing a user to create a set of specialized Viewports to his liking. Viewports also allow a set of *actions* to be added to its toolbar. When the user clicks an *action*, the Viewport forwards this information back to the connected application which can in turn act on the given *action*. Other features such as *auto focus* and *auto refresh* make the Viewport a very flexible widget. The drawing routines have been specifically designed to run in a separate thread space, interfering with the *emss* Core as little as possible.

```
// Initialize viewport
Viewport *viewport = new Viewport(100); // Refresh every 100ms
viewport->setAntiAlias(false);
```

```
// Register maps with viewport
viewport->registerMap(core->gridMap);
viewport->registerMap(core->physicalMap);
viewport->registerMap(core->terrainMap);
viewport->registerMap(core->heatMap);
viewport->registerMap(core->objectMap);

// Add toolbar actions
viewport->addToolbarAction("Task","task");
viewport->addToolbarAction("Nav Point","navpoint");
viewport->addToolbarAction("Environment Info","environmentinfo");
viewport->addToolbarAction("Find Robot","findrobot");

// Show widget and focus on robot
viewport->show();
viewport->focusOnPoint(core->robotObject->x, core->robotObject->y);

...

// Cleanup...
delete viewport;
```

Code 5-27: Viewport Widget Utilization Example

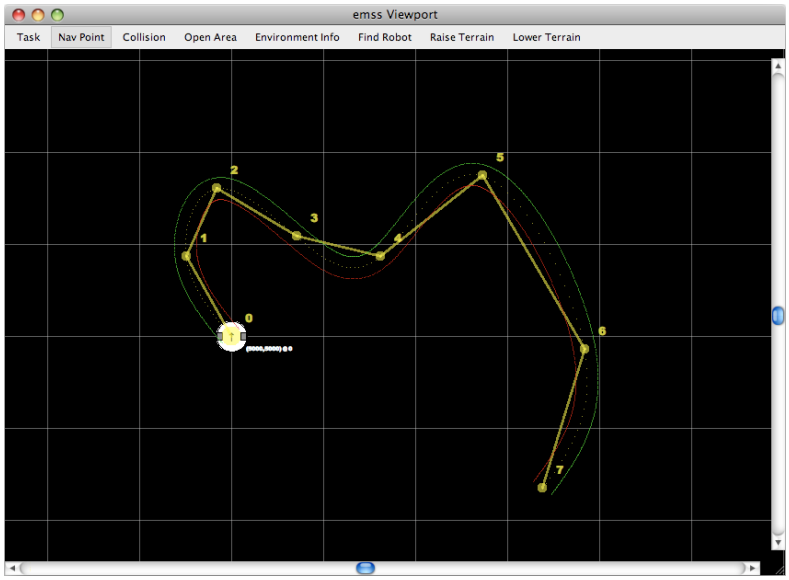


Figure 5-13: Viewport Widget

5.3.8.2 Joystick

The Joystick class can be used to control the various Controllers. It provides a widget which can be controlled by the mouse and provides all the appropriate slots and signals. The joystick has two yokes: *x* and *y*, which in turn can be used for control. Furthermore, it can be customized to force exclusive yokes – a mode where only one of the yokes can be moved at a time. In addition, the Joystick class also provides control over its acceleration and deceleration, a useful feature for gently controlling a robot.

5.3.8.3 Options Dialog

Often a user interface requires some sort of choice from the user. The Options Dialog class provides an easy to call, blocking static method *choose(options,description,title)* which creates a modal dialog of choices based on the *options* parameter. The choice in *options* must separated by the | character.

```
QString controller = OptionsDialog::choose("Block Drive|Fluid|emss", "Please
select a controller type:", "New Controller");
```

Code 5-28: Options Dialog Example

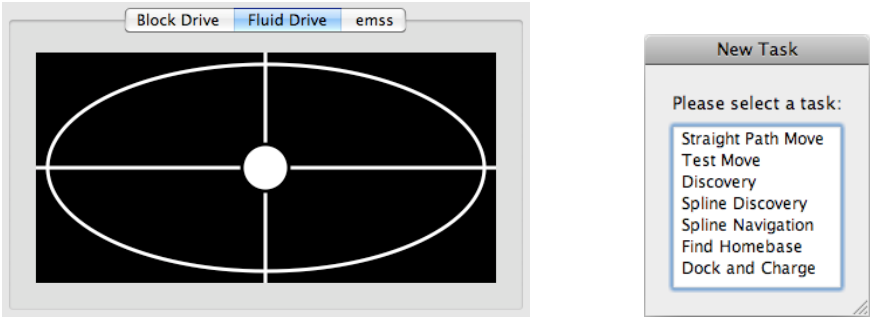


Figure 5-14: Joystick Widget (left), Options Dialog (right)

5.3.8.4 Settings Editor

Any emss application, including the Core itself, may have various settings. The Settings Editor class provides a simple text field where all settings are displayed and may be edited. The class accepts any standard *QSettings* data structure, making it plug-and-play with all *Qt* settings objects.

5.3.8.5 Task Editor

The Task Manager holds a list with all created Tasks. The Task Editor class allows an *emss* application to easily manage the Cores current Tasks. In addition to displaying Task Information, *Running* Tasks may be stopped.

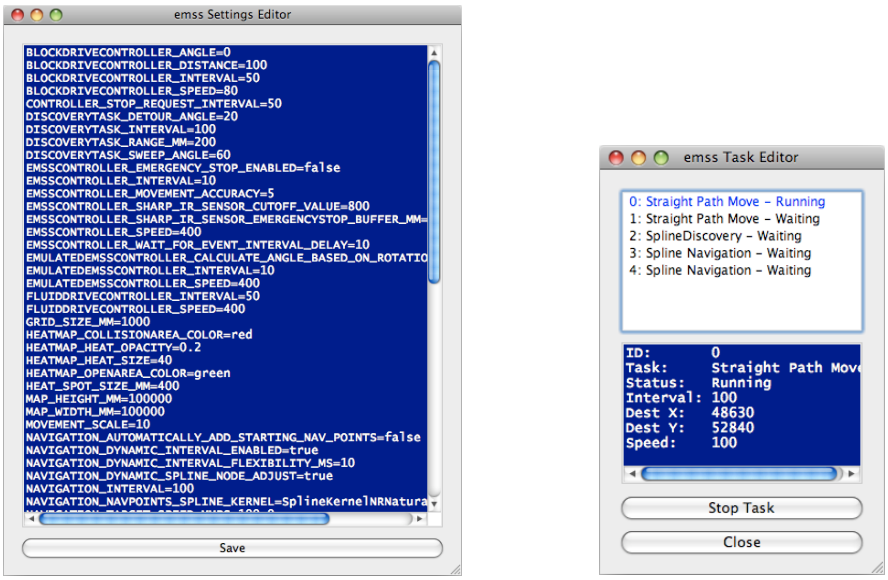
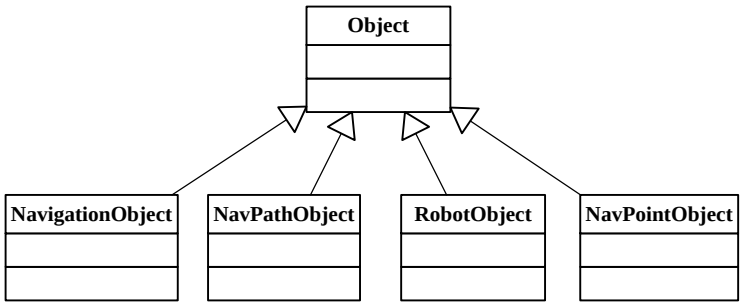


Figure 5-15: Settings Editor (left), Task Editor (right)

5.3.9 Objects



Core Objects are used to draw different structures on Maps, such as a navigation spline or the robots current position. Because of their trivial nature, they are not discussed in further detail here.

5.3.10 Core Configuration

Most of the Core components use many different constants to refine their algorithms or procedures. These settings are easily accessible through the Core methods *stringSetting(key), doubleSettings(key), boolSetting(key),* et cetera, and can be found in the resource file *emssCore.config*. The first advantage of using a configuration pool of variables opposed to using constants is the transparency of the Core configuration. As the *emss* Core grows, it becomes increasingly harder to keep and overview of all the configuration constants. Having all constants in one place lets one much easier see the overview and dependency between the many settings. The second advantage is that the settings can be adjusted at run time or before application start. In addition to the Core configuration, individual *emss* applications may also have configuration files for saving the internal state.

The following Core settings are currently available in the *emss* framework:

5.3.10.1 Controller

CONTROLLER_STOP_REQUEST_INTERVAL	Interval how long the controller should wait after a stop is requested

5.3.10.2 Block Drive Controller

BLOCKDRIVECONTROLLER_ANGLE	Default angle to turn
BLOCKDRIVECONTROLLER_DISTANCE	Default distance to move
BLOCKDRIVECONTROLLER_INTERVAL	Interval between commands
BLOCKDRIVECONTROLLER_SPEED	Default speed value

5.3.10.3 Fluid Drive Controller

FLUIDDRIVECONTROLLER_INTERVAL	Interval between commands
FLUIDDRIVECONTROLLER_SPEED	Default speed value

5.3.10.4 EMSS Controller

EMSSCONTROLLER_INTERVAL	Interval between commands
EMSSCONTROLLER_MOVEMENT_ACCURACY	Defines the variance from a value by movements

EMSSCONTROLLER_SHARP_IR_SENSOR_CUTOFF_VALUE	Defines the maximum distance of the sensor
EMSSCONTROLLER_SPEED	Movement speed
EMSSCONTROLLER_EMERGENCY_STOP_ENABLED	Defines if the robot should stop immediately or not
EMSSCONTROLLER_SHARP_IR_SENSOR_EMERGENCYSTOP_BUFFER_MM	Defines the distance where a stop is activate when a obstacle is detected
EMSSCONTROLLER_WAIT_FOR_EVENT_INTERVAL_DELAY	Defines the time the controller will wait

5.3.10.5 Emulated EMSS Controller

EMULATEDEMSSCONTROLLER_INTERVAL	Interval between commands
EMULATEDEMSSCONTROLLER_SPEED	Default speed value
EMULATEDEMSSCONTROLLER_CALCULATE_ANGLE_BASED_ON_ROTATIONAL_SPEED	The type of *getAngle()* COIL emulation

5.3.10.6 Discovery Task

DISCOVERYTASK_INTERVAL	Interval between commands
DISCOVERYTASK_DETOUR_ANGLE	Defines the angle to turn if an obstacle is detected
DISCOVERYTASK_RANGE_MM	Defines the distance ahead which should be scanned for obstacles
DISCOVERYTASK_SWEEP_ANGLE	Defines the angle to scan for obstacles

5.3.10.7 Spline Discovery Task

SPLINEDISCOVERYTASK_INTERVAL=100	Interval between commands
SPLINEDISCOVERYTASK_DETOUR_ANGLE	Defines the angle to turn if an obstacle is detected
SPLINEDISCOVERYTASK_EXPLORATION_MAP_GRID_SIZE	Defines the size of the exploration map
SPLINEDISCOVERYTASK_SWEEP_ANGLE	Defines the angle to scan for obstacles
SPLINEDISCOVERYTASK_RANGE_MM	Defines the distance ahead which should be scanned for obstacles
SPLINEDISCOVERYTASK_RELATIVE_EXPLORATION_HEIGHT_MM	Defines the height of the local exploration map. That means the robot is centered and the local map is built around it
SPLINEDISCOVERYTASK_RELATIVE_EXPLORATION_WIDTH_MM	Defines the width of the local exploration map. That means the robot is centered and the local map is built around it
SPLINEDISCOVERYTASK_TERRAIN_CUT_LEVEL	Defines the height of the terrain cut points
SPLINEDISCOVERYTASK_TERRAIN_MAP_MAX_HEIGHT	Defines the maximum height of the terrain points
SPLINEDISCOVERYTASK_TERRAIN_UNKNOWN_HEIGHT	Defines the value of a unknown height
SPLINEDISCOVERYTASK_TERRAIN_CUT_POINT_DISTANCE_MM	Defines the distance between two terrain cut points

5.3.10.8 Spline Navigation Task

SPLINENAVIGATIONTASK_INTERVAL	Interval between commands
SPLINENAVIGATIONTASK_RESET_NAV_POINTS_WHEN_DONE	Defines whether the navigation points were deleted after done (*true*) or not (*false*)

SPLINENAVIGATIONTASK_ALIGN_ROBOT_WITH_SPLINE	Defines whether the spline is aligned with the robot (*true*) or not (*false*)
SPLINENAVIGATIONTASK_CRITICAL_AREA_DISTANCE_MM	Defines at which distance an obstacle is in the critical area
SPLINENAVIGATIONTASK_ALIGN_ROBOT_WITH_SPLINE_SPEED	The speed at which the robot is aligned with the spline before beginning traversal

5.3.10.9 Test Move Task

TESTMOVETASK_INTERVAL	Interval between commands

5.3.10.10 Straight Path Move Task

STRAIGHTPATHMOVETASK_INTERVAL	Interval between commands

5.3.10.11 Task Manager

TASKMANAGER_INTERVAL	Interval between commands

5.3.10.12 Heat Map

HEATMAP_OPENAREA_COLOR	Defines the color of an open area object
HEATMAP_COLLISIONAREA_COLOR	Defines the color of a collision area object
HEATMAP_HEAT_OPACITY	Defines the opacity of a heat object in percent
HEATMAP_HEAT_SIZE	Defines the size of a heat object
REGISTERHEAT_OPACITY_COLLISION	Defines the opacity of a collision area object
REGISTERHEAT_OPACITY_OBJECT	Defines the opacity of a object
REGISTERHEAT_OPACITY_OPENAREA	Defines the opacity of a open area object
HEAT_SPOT_SIZE_MM	The size of a collision or open area heatspot

5.3.10.13 Physical Map

PHYSICALMAP_FILENAME	Defines the path to the resource for the physical map
PHYSICALMAP_SCALE	Defines the scale of the resource picture

5.3.10.14 Navigation

NAVIGATION_INTERVAL	Interval between commands
NAVIGATION_WHEEL_SPLINE_KERNEL	Defines which spline kernel is used for wheel spline calculation
NAVIGATION_NAVPOINTS_SPLINE_KERNEL	Defines which spline kernel is used for navigation point spline calculation
NAVIGATION_TARGET_SPEED_MMPS	Defines the target speed which is included by the spline calculation
NAVIGATION_DYNAMIC_INTERVAL_ENABLED	If *true*, a dynamic interval is allowed when querying *getWheelSpeed()*
NAVIGATION_DYNAMIC_INTERVAL_FLEXIBILITY_MS	The flexibility in milliseconds which causes the dynamic interval mode to rebuild the wheel splines
NAVIGATION_AUTOMATICALLY_ADD_STARTING_NAV_POINTS	If *true*, two start points are automatically added for a smoother spline start

NAVIGATION_DYNAMIC_SPLINE_NODE_ADJUST	If enabled (*true*), the navigational spline nodes are adjusted to match the current position of the robot, increasing the overall accuracy of the spline traversal

5.3.10.15 Robot

ROBOT_WHEEL_OFFSET_MM	Defines the distance between the wheel and robots middle point
ROBOT_SIZE_MM	Defines the radius of the robot
ROBOT_STARTING_POSITION_X_MM	Defines the start point of the robot in the map (*x* –coordinate)
ROBOT_STARTING_POSITION_Y_MM	Defines the start point of the robot in the map (*y* –coordinate)
ROBOT_BUMPER_COLLISION_OFFSET_MM	Defines the distance from the bumper to the middle point of the robot

5.3.10.16 General

MOVEMENT_SCALE	Defines the scale between the movement of the robot (given in mm precision) and the pixel drawing on screen.
GRID_SIZE_MM	Defines the grid size
MAP_HEIGHT_MM	Defines the height of the entire map
MAP_WIDTH_MM	Defines the width of the entire map
TERRAINMAP_RAISE_LOWER_INCREMENT=0.1	The default amount to increment, respective decrement, when raising or lowering the terrain map.
TERRAINMAP_RAISE_LOWER_SIZE=40	The default surface area size when raising or lowering the terrain map.

5.4 Applications

A few applications were built to test the theory and the control of the robot. Typically, it was helpful to implement an algorithm first in a so called test application before it was moved to the Core, as the smaller scope of a test application helped limit mistakes. Some of the test applications reference the Core library, while others are entirely independent. Our main graphical user interface to the *emss* Core is realized with the *emss* Interface application.

5.4.1 Interface

The Interface application *emssInterface* is the predominant graphical user interface to control the *emss* robot and other Core components. The application is divided into three groups: *Connection*, *Controller*, and *Data*. The Connection group allows one to connect and disconnect with the *emss* hardware. It supports the selection of all the implemented Core Controllers and Movement Trackers. The Controller panel features various tabs each designed specifically for the individual Controllers. The important diagnostic panel *History* is located inside the *Data* group box. The history widget provides all necessary diagnostic and debugging information which streams from the various Core components. Along with the *History* panel, other data sources are provided such as *Serial Port* and *Sensors*. The *emss* Interface allows multiple instances of Core Viewports, as well as modal windows

such as the Task Editor and Settings Editor of both the Core configuration settings and the Interface settings.

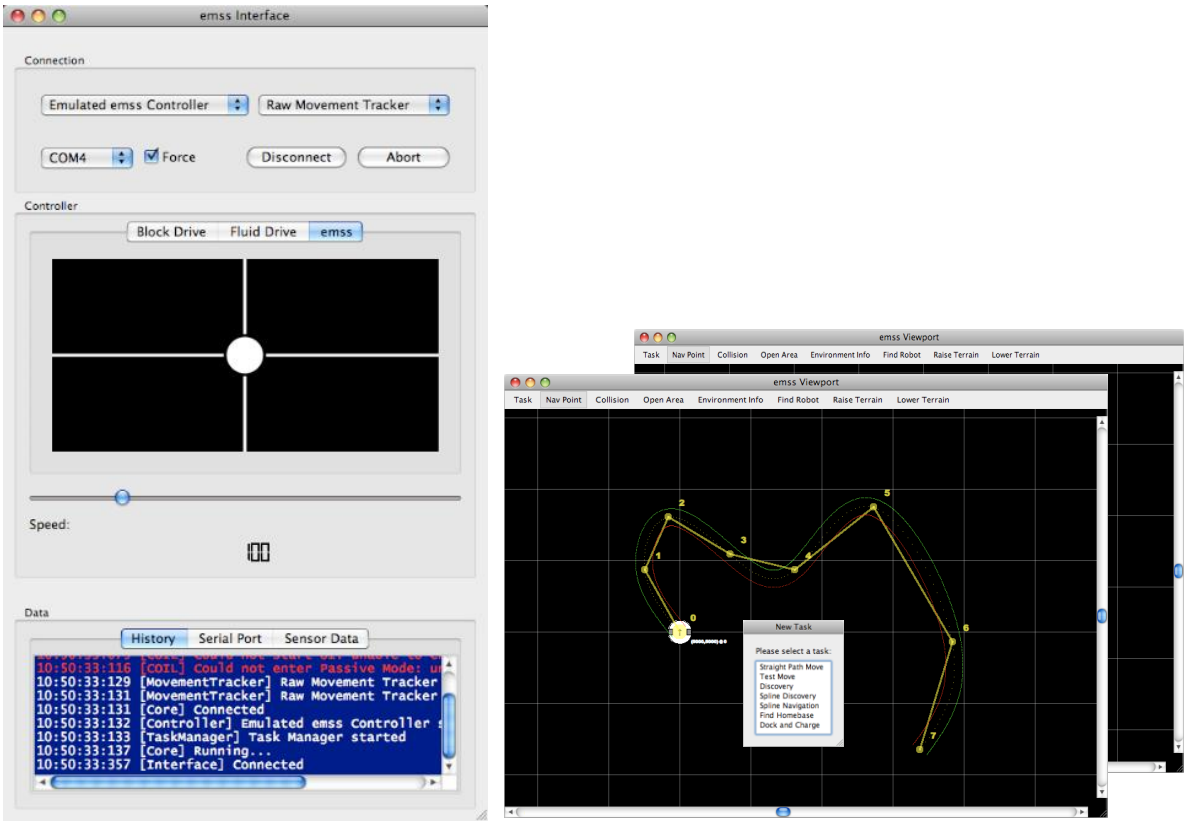


Figure 5-16: Interface with Multiple Viewports Screenshot

5.4.2 Simulation

Before the start of the project, and even the realization of the hardware, a simulation (*emssSimulation*) was created in order to get a head start on the algorithms, et cetera. Although never completed, many of the simulation components were ported to the *emss* Core when needed, including a large portion of the drawing code.

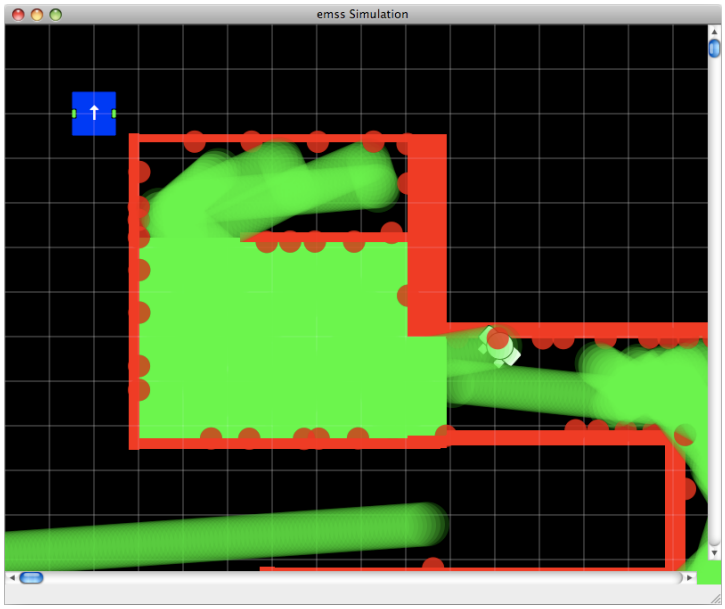


Figure 5-17: Simulation Screenshot

5.4.3 Vector Test

The Vector Test application (*emssVectorTest*) is a very simple program to visualize the rotation and translations of two vectors.

5.4.4 Rotation Test

The Rotation Test application (*emssRotationTest*) was built to help us to understand the concept of a rotation matrix. It is a very simple program with a modest GUI. This application holds two transformation matrices A_1 and A_2, and outputs the difference of angle and distance between them.

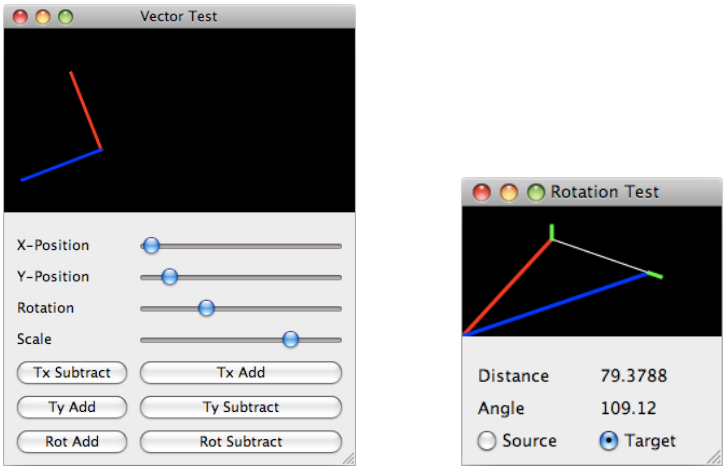


Figure 5-18: Vector Test Screenshot (left), Rotation Test Screenshot (right)

5.4.5 Spline Test

To test the different splines we implemented the *emssSplineTest* application. Various navigation points can be set, which in turn calculates the curve of the spline. Additional information such as the first derivative can be output as well.

5.4.6 Terrain Map

This application was built for testing the height map algorithm. It visualizes the exploration map together with a heat map. The red space is unexplored, the brighter the square the higher is the height map value. The two known obstacle and free space can be painted with the mouse. The yellow point represents the highest point in the map. It is also possible to show the cut points on different levels adjusted with a slider.

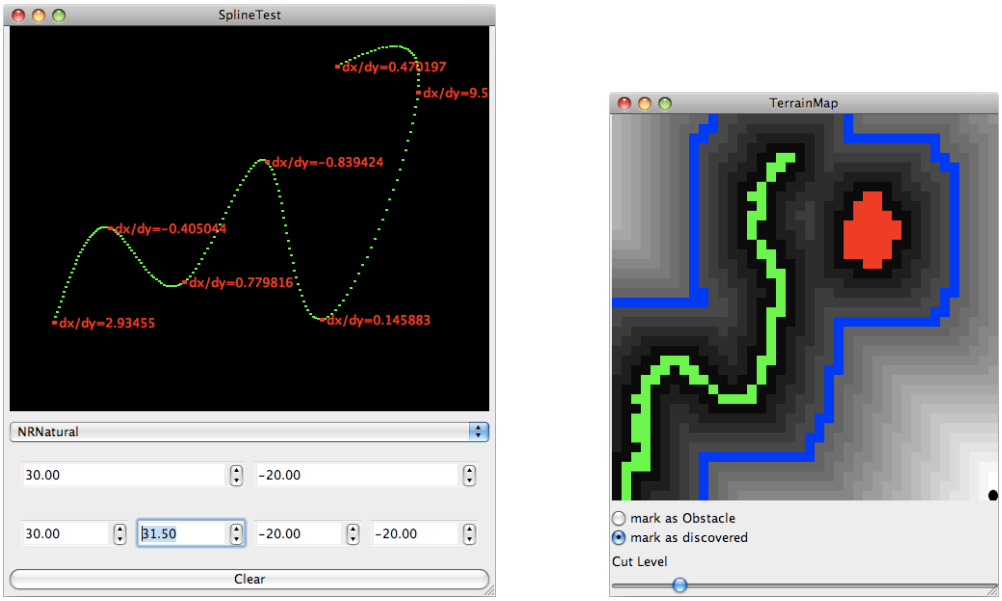


Figure 5-19: Spline Test Screenshot (left), Terrain Map Screenshot (right)

6 Project Report

6.1 Project Overview

6.1.1 Requirements

The goal of the *emss* project is to tackle the problems of a self-sustainable, environment mapping robot in a bottom up approach. The requirements of this approach include the realization of the following tasks:

R 1	Assembly of hardware
R 2	Communication between hardware and software
R 3	Robot control by human interaction
R 4	Robot control by navigational structures

Table 6-1: Requirements

An additional requirement which covers the entire project is the research and study of solutions to various problems presented by the different tasks. This requirement calls for the attention to theoretical problems in mathematical fields with application in robotics.

6.1.2 Planning

Because of the project's research nature, having no final destination, it was not possible to define concrete milestones and a fixed timeline. However, effort was taken to assume a course of path in order to keep the project in check during its lifetime. First, communication between hardware and software was planned, together with literature studies. Immediately following, basic maps were to be implemented. Once completed, the discovery algorithm was to be researched. The end of the project allowed for testing and documentation.

Week 1	Buy and assemble hardware, first tests
Week 2	Workspace setup, port COIL to Qt, accuracy tests
Week 3	Literature studies, COIL abstraction layer
Week 4	Literature studies, build Simple Controller
Week 5	Collision and Obstacle Map
Week 6	Heat Map Implementation
Week 7	Discovery Algorithm
Week 8	Further research
Week 9	Further research
Week 10	Further research
Week 11	Testing, bug fixing
Week 12	Documentation
Week 13	Documentation
Week 14	Finish up documentation, reserve time

Table 6-2: Proposed Project Timeline

6.1.3 Logbook

Throughout the course of the project a detailed log book was kept. As there were multiple actors playing a role in the design and implementation of *emss*, it was important that a majority of the information was available at any time to all project members. In addition, it proved interesting and useful to have a central collection of ideas and actions taken over the course of the project. If one wanted to seek forgotten ideas or problems, or even solutions, the logbook was consulted. The following table lists the topics discussed in the logbook for every week.

Week 1	Bought the hardware Mounted the hardware First tests
Week 2	Set up the workspace Ported COIL to Qt Accuracy tests
Week 3	Literature Studies Began Documentation Started Simple Controller GUI and main functions BlockDriveController implementation Wrote SensorTest for receiving and display the various sensor values Created joystick widget for directly control the robot Created the core project
Week 4	FluidController implementation Restructuring of Core for new architecture decision Added additional function to the Simple Controller Added ViewPort to the Simple Controller
Week 5	Tasks added to Core ViewPort was extended with actions First implementation of the emssController Sharp IR Finder was mounted on the top of the robot Sensor data from Sharp IR finder are handled in the Core
Week 6	Heat Map was created Simple implementation of a discovery task
Week 7	Restructure of Core after new design decisions EmssController was extended with the EmulatedEmssController to emulate the robot Practice of rotation matrix theory and wrote the RotationTest
Week 8	Implementation of the rotation matrix in the Core Added more test tasks Research of splines and how to use them Wrote the test application SplineTest
Week 9	Implementation of C2 Splines Change to use a spline for each wheel Navigation added
Week 10	Research in navigation in an unknown area Built TerrainMap to test a height map based algorithm
Week 11	Implemented SplineDiscoveryTask using TerrainMap Added terrain cut points and a sort algorithm to the terrain map to improve the exploration
Week 12	Added a settings editor Implementation of the terrain map Changed wheel spline navigation in the Navigation
Week 13	(Documentation)
Week 14	(Documentation)

Table 6-3: Logbook Topics

6.1.4 Milestones

In a project that requires research to accomplish its goals it is impossible to define detailed milestones. Therefore we defined global milestones which describe which functionality at a specific time point should be implemented in the system:

Week 1	Hardware	All robots hardware parts are mounted
Week 5	Robot control by human	The robot can controlled by a human remotely
Week 9	Robot control by computer	The robot can controlled by tasks, the only thing a human has to do is to define tasks which were processed
Week 11	Code freeze	No more changes were made in code
Week 14	Project end	Finish documentation and release

Table 6-4: Project Milestones

6.1.5 Organization

6.1.5.1 Persons

Name	**Mail**	**Role**
Daniel Krüsi	daniel.kruesi@hsr.ch	Project member
David Grob	david.grob@hsr.ch	Project member
Prof. Dr. J. Wirth	joachim.wirth@hsr.ch	Adviser and evaluator
Prof. S. Keller	stefan.keller@hsr.ch	Evaluator

Table 6-5: Project Personal

6.1.5.2 Weekly Meetings

Each week on Tuesday we held a meeting with our advisor. In these meetings we talked about the work we have done the previous week, as well as discussed the mathematical, conceptual, and software problems in our project.

6.1.5.3 Code Versioning

The code was managed with *Tigris* Subversion. This allows us to view or to go back to any committed version of the code. The SVN server *svn.nerves.ch* which hosts our data is backed up nightly and serves as our backup system.

6.1.5.4 Code Comments

Comments throughout our code are well maintained. Self-explanatory methods were omitted from comments in order to prevent bloated files. In the *Eclipse* IDE we defined three comment tags (below), which are compiled into a visual list during compilation:

- IDEA: A possible approach how something could be implemented
- TODO: Things not implemented yet, minor corrections to change
- BUG: An unsolved problem (bug) in the code

6.1.5.5 Programming Tools

Software	Producer	Version
Eclipse	*Apache Software Foundation*	3.4.0
Eclipse C/C++ Development Tools	*Apache Software Foundation*	5.0.0
Qt C++ Eclipse Integration	*Trolltech*	1.4.1
Qt	*Trolltech*	4.4.1

Table 6-6: Programming Tools

6.1.5.6 Metrics

The total SLOC (not including comments or white spaces) for the entire *emss* project is 10,138 lines, spanning 158 files. The *emss* Core totals 6,554 SLOC, with 90 files.

6.2 Personal Experience Report

6.2.1 Daniel Krüsi

Ever since I was a young child, robots seemed to be a naturally instilled fascination. As I grew older, and found a deep interest in computer science, building a self-sustainable robot has been a dream of mine. The *emss* project has let me realize part of this dream, which in turn has brought me great joy. The software challenges presented by this project were very interesting, and provided a healthy environment for trying, failing, learning, and succeeding. Furthermore, the investigation of problems using theoretical solutions has strengthened my value in mathematics, and taught me priceless lessons of how to research and apply theory. Working with David Grob and Joachim Wirth was a great experience, and I wouldn't have wanted it any other way.

6.2.2 David Grob

Eine selbst ausgewählte und eingereichte Arbeit hat den Vorteil dass man mit Bestimmtheit ein interessantes Thema bearbeitet. Mich fasziniert das Gebiet der Robotik stark. Desto fest freute ich mich ein solches Projekt durchzuführen und mich intensiv mit dem Thema zu befassen. Mit Daniel habe ich einen ebenso interessierten Projektpartner gefunden. Diesen Enthusiasmus teilte auch unser Betreuer Prof. J. Wirth was auch sehr zu unserem guten Klima beigetragen hat.

Projektverlauf

Der Projektstart verlief planmässig. Als erstes musste ein erhebliches Studium von Literatur durchgeführt werden um einen Einblick in die Thematik zu erhalten. Es stellte sich sehr schnell heraus, dass unzählige Lösungswege und Theorien existieren welche zum Ziel führen. Deshalb galt es ein wenig zu selektionieren und zu entscheiden welche Ideen wirklich zur Erreichung unserer Anforderungen hilfreich sein könnten. Nach dem Literaturstudium folgte die Implementierung unserer Ideen aufbauend von der Ansteuerung des Roboters mittels der seriellen Schnittstelle über das Lesen der Sensordaten bis hin zur komplexeren Berechnung der Fahrwege und Erkundungsalgorithmen. Nach dieser Implementationsphase mussten wir stoppen und unsere Dokumentation vervollständigen. Wir konnten leider nicht alle Idee implementieren welche wir vor dem Projektstart definiert hatten. Dies hat den Grund, dass wir den Aufwand etwas unterschätzt haben. Wir wollten eine möglichst

schöne Lösung deshalb haben wir uns auch dafür entschieden Splines zu implementieren und nicht auf ein Stop-and-Go-Verfahren zu setzen. Zu dem kommt dazu dass ein Teilgebiet wie Navigation oder Erkundungsalgorithmen Arbeit für eine eigene Studienarbeiten bietet.

Herausforderungen

Für mich waren die grössten Herausforderungen mathematischer und sprachlicher Natur. Der Mathematikunterricht hier an der Fachhochschule bietet zwar einige Grundlagen, behandelt aber einige Themen zu wenig intensiv. Vor allem Lineare Algebra wird meiner Meinung nach zu wenig stark gewichtet. Aus diesem Grund mussten wir viel Mathematik mühsam selbstständig erlernen. Wir haben uns auf eine englischsprachige Dokumentation geeignet, dies bedeutet für uns beide Mehrarbeit. Da meine Englischkenntnisse nur minimal ausreichen um so eine Arbeit zu schreiben und ich mehr Zeit benötige um ein Satz verständlich zu formulieren. Daniel musste alle Texte nochmals überarbeiten und verbessern.

Fazit

Das Projekt war sehr interessant und ich habe viel dazu gelernt. Themen wie Navigationsalgorithmen, Erkundungstechniken, Lineare Algebra vertiefte ich sowie auch meinen Englischwortschatz konnte ich erweitern. Die Zusammenarbeit mit Daniel verlief einwandfrei, wir ergänzten uns optimal und hatten die gleichen Vorstellungen vom Projekt. Etwas Schade fand ich, dass wir die meisten Themen nur sehr kurz behandeln konnten. Ich freue mich deshalb bereits auf die Folgearbeit in der wir uns (hoffentlich) nur auf eine Problematik konzentrieren können.

7 Appendix

7.1 Glossary

Auxiliary Tower: Serves a platform for adding additional sensory equipment and other accessories.

Block Drive Controller: A simple controller to move a fixed distance forwards or backwards, or to turn a predefined angle.

Bumper: A buffer bar to recognize collisions in front of the robot.

CCW: Counter-clock-wise.

Charging Station: The charging station provides the recharging hardware for both the robot and laptop battery.

Cliff Sensors: Downward-pointing sensors disabling the robot from driving down stairs or other sudden drops.

COIL (Create Open Interface Library): This library opens a serial port and directly communicates with the *iRobot*.

Collision Area: Represents any sort of collision or obstacle, forming the environments boundary.

Controller: Sends movement commands to COIL and is also responsible for passing along sensor data to other Core components. The Controller is also a base class of fully functional Controllers.

Cut Level: Is used in the height map to cut the terrain at specific height.

Cut Level Points: Points in the map with the exact height of a cut level.

Differential steering: If each wheel is controlled independently of the other.

Discovery Task: A Task with the goal to explore an unknown area.

Emss Controller: The main Controller of the *emss* software.

***emss* (Environment-Mapping Self Sustainable) Robot**: A robot which interacts autonomously and explores an unknown area. It realizes if its job is finished or the battery status is low and reacts correspondingly.

***emss* Interface**: A graphical interface to control the robot and create Tasks.

Emulated Emss Controller: Emulates the *emss* Controller so that no physical robot is needed for executing tests.

Environment mapping: Building a map representing the environment using sensor data.

Exploration Map: Defines a map with all the aquired details about the environment.

Fluid Drive Controller: Supports control of the robot with a two-dimensional GUI joystick.

Framework: A framework is an extensible structure for describing a set of concepts, methods and technologies.

Heat Map: A Heat Map is built to visualize the discovered environment of the robot with different colors.

Height Map: A map which visualizes a terrain.

IR Range Finder: A device used for detected distance objects using infra-red waves.

iRobot Create: The *iRobot Create* is a robust, rugged and versatile robot base kit that can be used for various robotics hobby and research applications.

Local Height Map: A local height map is a fixed size map with the robot in its center. It is used by the height map algorithm.

Map: A visual representation of an environment.

Movement Tracker: Responsible for tracking movements of the robot, and in turn performing the localization, a Movement Tracker accepts signals from the active Controller and translates them accordingly.

Navigation: The Navigation class holds the information of the navigation points, or way points, for the robot. Other data structures, such as splines, are also included in this class.

Open Area: No collision is at this point and portrays the robots path through the environment.

Physical Map: The *emss* Physical Map represents the real world environment in its physical state.

***Qt* Framework**: *Qt* is a cross-platform application framework produced by *Trolltech*.

Raw Movement Tracker: The Raw Movement Tracker tracks the robot's movement by geometrically interpreting the sensor data sent back from the robot.

Self-sustainable: The ability to provide one's own resources and needs without external help.

SLOC: Source Lines of Code.

Spline Discovery Task: Implementation of a discovery algorithm using splines and a height map.

Spline Navigation Task: A navigation Task using splines for smooth movement.

Spline: Smooth, piecewise defined function of polynomials.

Straight Path Move Task: This task moves the robot in a linear fashion to the given destination.

Task Manager: Task Manager receives a Task, appends it to the Task List and starts to execute it.

Terrain Map: Visualizes the exploration map together with a heat map with the aim of calculating a navigational terrain.

Test Move Task: Includes test movements. These movements are used for diagnostic purposes, such as calibration, accuracy observation, and general research.

Transformation matrix: A description of the position and alignment of an object.

Vector: Objects in a vector space.

Vector space: Vector space is a set of objects (called vectors) that can be scaled and added.

VGA: A resolution of 640x480.

Viewport: A Viewport has the ability to display any *emss* Map as well as the Maps Objects.

7.2 List of Figures

7.3 List of Code Segments

7.4 List of Tables

7.5 List of Hardware

- *iRobot Create*
- *iRobot* Homebase and Charging Station
- *iRobot* Advanced Battery Supply
- *iRobot* Serial Connection Cable
- USB to Serial Port Adapter
- *Sharp* IR Range Finder
- *Logitech* QuickCam
- *Averatec* 2100 Laptop
- *D-Link* Wireless b/g USB Adapter
- USB Extension Cable
- USB 2.0 Multiport Adapter

7.6 Measurement Data

Speed (mm/s)	Distance Requested (mm)	Distance Reported (mm)	Actual Distance (mm)	Error (mm)	Error (%)	Actual Error (mm)	Actual Error (%)
20	100	103	111	3	2.91262	8	7.2072
20	200	200	210	0	0	10	4.7619
20	400	403	416	3	0.74442	13	3.125
20	500	504	519	4	0.79365	15	2.8902
20	1000	1002	1025	2	0.1996	23	2.2439
40	100	111	122	11	9.90991	11	9.0164
40	200	207	223	7	3.38164	16	7.1749
40	400	409	430	9	2.20049	21	4.8837
40	500	506	528	6	1.18577	22	4.1667
40	1000	1009	1036	9	0.89197	27	2.6062
60	100	118	140	18	15.2542	22	15.714
60	200	201	229	1	0.49751	28	12.227
60	400	411	438	11	2.6764	27	6.1644
60	500	515	544	15	2.91262	29	5.3309
60	1000	1019	1057	19	1.86457	38	3.5951
80	100	104	132	4	3.84615	28	21.212
80	200	215	240	15	6.97674	25	10.417
80	400	411	430	11	2.6764	19	4.4186
80	500	522	539	22	4.21456	17	3.154
80	1000	1026	1030	26	2.53411	4	0.3883
100	100	130	163	30	23.0769	33	20.245
100	200	230	260	30	13.0435	30	11.538
100	400	401	428	1	0.24938	27	6.3084
100	500	502	528	2	0.39841	26	4.9242
100	1000	1010	1031	10	0.9901	21	2.0369
150	100	151	205	51	33.7748	54	26.341
150	200	203	254	3	1.47783	51	20.079
150	400	412	461	12	2.91262	49	10.629
150	500	513	564	13	2.53411	51	9.0426
150	1000	1033	1082	33	3.19458	49	4.5287
300	100	106	318	6	5.66038	212	66.667
300	200	214	312	14	6.54206	98	31.41
300	400	429	535	29	6.75991	106	19.813
300	500	535	643	35	6.54206	108	16.796
300	1000	1070	1175	70	6.54206	105	8.9362

Table 7-1: *iRobot Create* Movement Accuracy

Distance Measured (mm)	Sensor Offset (mm)	Actual Distance (mm)	Analog Signal	Reverse Analog	Ratio
0	75	75	542	458	0.1638
20	75	95	534	466	0.2039
40	75	115	466	534	0.2154
50	75	125	429	571	0.2189
60	75	135	396	604	0.2235
80	75	155	354	646	0.2399
100	75	175	320	680	0.2574
150	75	225	254	746	0.3016
200	75	275	212	788	0.349
250	75	325	183	817	0.3978
300	75	375	162	838	0.4475
350	75	425	146	854	0.4977
400	75	475	126	874	0.5435
500	75	575	110	890	0.6461
600	75	675	98	902	0.7483
700	75	775	90	910	0.8516
800	75	875	82	918	0.9532
900	75	975	78	922	1.0575
1000	75	1075	75	925	1.1622

Table 7-2: IR Range Finder Signal to Millimeter Conversion Data

7.7 Installation and Build Instructions

In order to successfully build the *emss* framework, the following requirements must be met:

- *GNU* C++ Compiler, version 3.0 or newer
- *Qt* Open Source Edition for C++, version 4.4 or newer

The *emss* software requires only the standard *Qt* build. No additional modules are required as everything is packaged within the *emss* source. Additionally, the *Qt* debug build is not required as long as *emss* software is linked against *Qt* with release binaries.

All *emss* binaries are compiled to either */emss/Binaries/Unix* and */emss/Binaries/Windows*, depending on which environment the build is executed in.

To build the *emss* Core library, which is required for *emss* applications, the following commands can be used:

```
cd <path>/emms/Core
qmake
make
```

To build any *emss* application, such as the *emss* Interface, the following commands can be used:

```
cd <path>/emms/<application directory>
qmake
make
```

The following application directories are available for building:

- Interface
- COILTest
- RotationTest
- SensorTest
- Simulation
- SplineTest
- TerrainMap
- VectorTest

In order to execute any *emss* application, the following dependencies must be resolved (i.e. libraries in the *path*):

- emssCore
- QtCore4
- QtGui4

When first starting an *emss* application which requires a configuration file, the path to the wanted configuration file is requested. Pre-defined configuration files can be found at */emss/Resources/emss<configuration>.config*, such as */emss/Resources/emssCore.config*.

7.8 References

Bartels, R. H., Beatty, J. C., & and Barsky, B. A. (1998). *An Introduction to Splines for Use in Computer Graphics and Geometric Modelling.* San Francisco: Morgan Kaufmann.

Boor, C. d. (1978). *A Practical Guide to Splines.* Springer-Verlag.

Čapek, K. (1921). *Rossum's Universal Robots.* Prague.

Catmull, E. a. (1974). *Computer Aided Geometric Design.* New York: Eds. Academic Press.

Chan, C. (n.d.). *Averatec 2100 Series Review*. Retrieved 2008, from Notebook Review: http://www.notebookreview.com/default.asp?newsID=2991

Conte, S., & deBoor, C. (1972). *Elementary Numerical Analysis.* New York: McGraw-Hill.

Fowler, C. B. (1967). The Museum of Music: A History of Mechanical Instruments. *Music Educators Journal* .

Handy, J. (Director). (1940). *Leave It To Roll-Oh* [Motion Picture].

iRobot Corporation. (2006). *iRobot® Create Open Interface.*

iRobot Corporation. (2006). *iRobot® Create Owners Manual.*

Jonathan, K., Jonathan, B., Arvind, P., Omair, K., & Sukhatme, G. S. (2008). *Just Add Wheels: Leveraging Commodity Laptop Hardware for Robotics and AI Education.* Los Angeles: University of Southern California.

Kelly, J., Binney, J., Pereira, A., Khan, O., & Sukhatme, G. S. (2008). *Just Add Wheels: Leveraging Commodity Laptop Hardware for Robotics and AI Education.* Los Angeles: University of Southern California.

Kostandov, M., Schwertfeger, J., & Jenkins, O. C. *Toward Real-time Human Detection and Tracking in Diverse.* Providence: Brown University.

Kostandov, M., Schwertfeger, J., Jenkins, O. C., Jianu, R., Buller, M., Hartmann, D., et al. (2007). *Robot Gaming and Learning using Augmented Reality.* Brown University.

Krusi, D., & Grob, D. (2008). *Environment Mapping Self-Sustainable Robot Proposal.* Zurich.

Krusi, S. (2008). *Modeling Forward Trajectories of a Robot.* College Station: Texas A&M University.

Logitech. (2006). *QuickCam Connect Manual.*

Martinez, D. M., Haverinen, J., & Roning, J. (2008). *Sensor and Connectivity Board (SCB) for Mobile.* Oulu: University of Oulu.

Martınez, D. M., Haverinen, J., & Roning, J. (2008). *Sensor and Connectivity Board (SCB) for Mobile Robots.* Oulu.

Mataric, M. J., Koenig, N., & Feil-Seifer, D. (2007). *Materials for Enabling Hands-On Robotics.* Los Angeles.

Nathan, K. (2007). *Toward Real-time Human Detection and Tracking in Diverse.* iRobot Corporation.

Olson, L. (2008). *Cubic Splines, Bezier Curves.* Urbana-Champaign: University of Illinois.

Runge, C. (1901). Über empirische Funktionen und die Interpolation zwischen äquidistanten Ordinaten. *Zeitschrift für Mathematik und Physik* .

Sharp. (2002). *Sharp GP2D12 IR Range Finder.* Sharp Publishing.

Takacs, B., & Hanak, D. (2008). *Home Robots and Ambient Facial Interfaces.* Budapest.

Thorpe, C., Clatz, O., Duggins, D., Gowdy, J., MacLachlan, R., & Ryan, J. (2001). *Dependable Perception for Robots.* Carnegie Mellon University.

Weisstein, E. W. (1999). Vector. *MathWorld* .

Wood, G. (2002). *Living Dolls: A Magical History Of The Quest For Mechanical Life.*

Yen-Chun, L., Yen-Ting, C., Szu-Yin, L., & Jen-Hua, W. (2008). *Visual Tag-based Location-Aware System for Household Robots.*