Technical documentation

Editor: Marcus Bäck

Version 1.0

Status

Reviewed	Marcus Bäck	2014-12-11
Approved	Hanna Nyqvist	2014-12-11

PROJECT IDENTITY

2014/HT, Invenire Periculosa
Linköping University, Dept. of Electrical Engineering (ISY)

Group members

Name	**Responsibility**	**Phone**	**Email** (@student.liu.se)
Martin Szilassy (MS)	Project manager (PM)	070-8840295	marsz918
Marcus Bäck (MB)	Responsible for documentation (DOC)	070-6924804	marba751
Victoria Alsén (VA)	Chief of tests	073-9409540	vical845
Johan Källström (JK)	Chief of design	073-0718371	johka546
Mikael Hammar (MH)	Master of SLAM	070-2658294	mikha087
Daniel Örn (DÖ)	Chief of hardware	073-6448910	danor434
Olof Zetterlund (OZ)	Chief of information	073-7133822	oloze183
Simon Ollander (SO)	Junior technical advisor	073-8322898	simol515

Email list for the whole group: balrog-2014@googlegroups.com
Web site: `http://www.isy.liu.se/edu/projekt/tsrt10/2014/bandvagn/`

Customer: SAAB Bofors Dynamics, Linköping
Customer contact: Torbjörn Crona, torbjorn.crona@saabgroup.com
Course leader: Daniel Axehill, +46 13 284042, daniel@isy.liu.se
Client: Hanna Nyqvist, +46 13 281 353, hanna.nyqvist@liu.se
Tutor: Martin Lindfors, +46 13 281365, martin.lindfors@liu.se

Contents

Document history

Version	**Date**	**Changes**	**Sign**	**Reviewed**
0.1	2014-12-05	First draft	IP	MB
0.2	2014-12-09	Lots of changes. Updated according to comments from client and tutor.	IP	MB
0.3	2014-12-10	Corrections according to client and tutor	IP	MB
1.0	2014-12-11	Version 0.3 approved by client	MS	MB

1 Introduction

Balrog is an ongoing project to construct a mine sweeping crawler robot. It is a joint project between Saab Bofors Dynamics and Linköping University.

The project was started in the spring of 2009 with the student group O'hara's. O'hara's main focus were specifications, control programs, network communication and navigation techniques.

In autumn 2009 the group Carpe Locus continued where O'hara's had left and developed remote control of the crawler and automatic control for the propulsion. The crawler was also equipped with GPS.

The next autumn (2010) the group 8Yare continued by mounting an industrial computer on the crawler and ported the old code to the new computer. The crawler was also mounted with stereo cameras (model Bumblebee 2).

In 2011 group iMAP focused on using the camera and the crawlers sensors to create a 3D map of the operating area. In 2012 Minenmarker choose not to continue the work with the 3D map. Instead they developed the mine sweeping functionality and improved the crawlers positioning. The main goal of the project in 2012 was to verify that the entire area had been searched and that all the mines were found. Minenmarker also named the robot Balrog.

In 2013 the group Ostende Abscondita continued to improve the positioning and search algorithm. They also integrated a wireless hand controller for manual control of Balrog.

This year, 2014, the main objectives of the project were to continue to develop Balrog's positioning. Balrog was equipped with a 360° laser scanner and a more powerful industrial computer.

The technical documentation is a detailed overview of how the system is designed and implemented. The purpose of this document is to provide the client and customer with a thorough description of the system that has been implemented. It also serves as a detailed description of the implementation and what is left, for the future project groups that will continue the work with Balrog. All the requirements of the functionality of Balrog is found in the Requirement specification [1].

2 System overview

The autonomous mine sweeping system consists of three main subsystems; the base station, the hand controller and Balrog. Balrog is a tracked vehicle shown in Figure 2.1 which is divided into smaller subsystems, see Section 6. For a schematic overview of the system see Figure 2.2.

Figure 2.1: The final assembly of Balrog 2014.

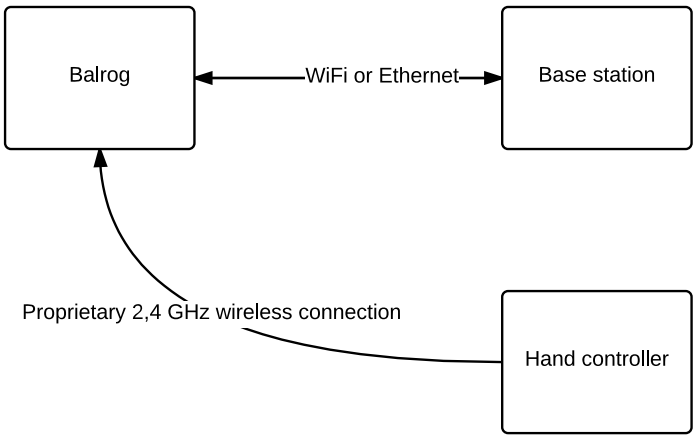

Figure 2.2: Illustration of the different parts in the system and how they interact.

2.1 Coordinate frames

The system uses four different coordinate frames.

2.1.1 Global coordinate frame

The coordinates (x^g, y^g, z^g) are defined in the global coordinate system. The global coordinate system is given according to the RT90 standard which is a geodetic system adapted for Sweden with origin corresponding to the WGS84-coordinate N0°0.000′ E0°0.000′ at sea level. x^g is the longitude, y^g is the latitude and z^g is the altitude above sea level for the position (x^g, y^g).

2.1.2 Local coordinate frame

The coordinates (x^l, y^l, z^l) are defined in the local coordinate frame, a translated and rotated version of the global frame as shown in Figure 2.3. The origin of the local frame is always located in the bottom left corner in the area of operation. The local frame is a right hand system with x^l and y^l making out the two dimensional internal map in the system, pointing right and upwards respectively, and z^l pointing outwards of the map.

In order to express global coordinates in the local frame we need to know the translation and rotation. If the global coordinates of the origin of the local system (bottom left corner of the operational area) (x^g_{bl}, y^g_{bl}) and the global coordinates of the bottom right corner of the operational area (x^g_{br}, y^g_{br}) are known, the rotation α and then the transformation can be computed. Since the z coordinate is not used for our application it will be left out of the transformation.

$$\alpha = \text{atan2}\left(y^g_{bl} - y^g_{br}, x^g_{bl} - x^b_{br}\right) \tag{2.1}$$

$$\begin{bmatrix} x^l \\ y^l \end{bmatrix} = \begin{bmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} x^g - x^g_{bl} \\ y^g - y^g_{bl} \end{bmatrix} \tag{2.2}$$

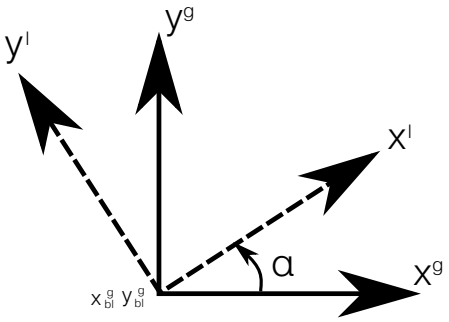

Figure 2.3: Global and local frame relationship.

2.1.3 Balrog coordinate frame

The coordinates (x^b, y^b, z^b) are defined in the Balrog coordinate frame and are fixed relative to Balrog. x^b points forward, y^g to the left and z^g upwards. The origin is in the

center of Balrog. The rotation around the local frame is defined as the angle ψ as is shown in Figure 2.4.

If the origin of the Balrog frame has the coordinates (x^l_{b0}, y^l_{b0}) in the local frame, the transformation can be computed as shown in equation 2.3

$$\begin{bmatrix} x^b \\ y^b \end{bmatrix} = \begin{bmatrix} \cos\psi & \sin\psi \\ -\sin\psi & \cos\psi \end{bmatrix} \begin{bmatrix} x^l - x^l_{b0} \\ y^l - y^l_{b0} \end{bmatrix} \tag{2.3}$$

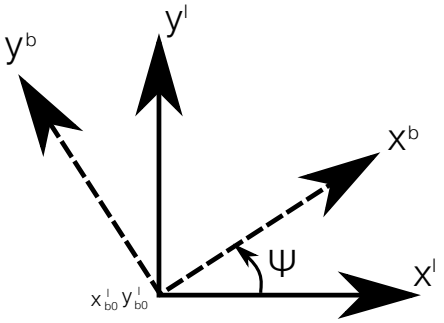

Figure 2.4: Balrog and local frame relationship.

2.1.4 Grid coordinates

The obstacle and probability map uses a gridded coordinates due to the discrete representation of the map. This coordinate frame is also used by the route planning algorithm. A grid coordinate is calculated by rounding down a local position coordinate in regard of the resolution of the grid map. When the map is created the resolution of the grid coordinate is specified with number of grids per meter. When transforming a grid coordinate to a local position the resulting position is given in the centre of the grid. See Section 3.7 for more details.

3 Implementation

This section contains information about the implementation of the functionality, the development and testing environment and the overall program structure and flow.

3.1 Overview

- Programming language: `C++11`
- Compiler: gcc 4.6.3 [2]
- Target platform: Ubuntu 14.04.1 LTS (Trusty Tahr) [3]
- Makefile generator: qmake 2.01a [4]
- Makefile generator: cmake 2.8.12.2 (used when building third party libraries) [5]
- Makefile generator: autoconf 2.69 (used when building third party libraries) [6]
- Integrated development environment (IDE): Qt Creator 3.1.1 [7]
- Version control: Git [8]

3.2 Dependencies

- Logging library: google-log 0.3.3 [9]
- Testing library: google-test 1.7.0 [10]
- Matrix library: Eigen 3.2.2 [11]
- Threading library: pThreads [7]
- Laser sensor drivers: RPLIDAR development kit 1.0.1 [12]
- Build essentials package for Ubuntu 14.04 [13]

For a step by step user installation guide please see the user manual [14].

3.3 Unit tests

The project has used unit tests during the development. A unit test is a black box test, where a function is tested whether it returns the expected value when a specific argument is passed to the function. The unit tests does not have complete coverage, but it is an easy way of checking if the functionality behaves as intended after a new implementation. The unit tests were executed when trying to push to the master branch on Git. The convention that the master branch must build and execute all tests at any time was used.

Two different unit test projects were created. One that was independent of hardware and another that needed hardware to execute. The automatic build system Travis CI [15] was used for automatic building and test execution, but since Travis had no access to our hardware the unit tests requiring hardware could not automatically be tested since they will always fail if hardware is not present.

3.4 Projects

The code is split into 7 different units called projects, listed in Table 3.1. Each project can be compiled individually, but some may depend on headers and lib files produced by other projects.

Table 3.1: Table of projects.

Project name	Type	Description	Dependencies
`common`	static library	Contains implementation shared between the basestation and Balrog	-
`common2013`	static library	Legacy code from previous years project	-
`balrog`	static library	Code intended to run on Balrog	`common`
`basestation`	executable	Basestation software	`common`, `common2013`
`app_balrog`	executable	Application run on Balrog	`common`, `balrog`
`tests`	executable	Contains unit tests for functionality that does not need hardware	`common`, `balrog`
`tests_hardware`	executable	Contains unit tests for functionality that uses hardware	`common`, `balrog`

3.5 Shared data structure

Shared data structures holds common data in the system. The data is shared between different subsystems by passing in a reference to the data when the subsystem is instantiated. All data manipulation is implemented to be safe for concurrency, race condition etc. In practice this mean mutually exclusive locks in pThreads, but the objects using the shared data structures can be ignorant of this and use them as if all manipulating tasks were atomic.

3.5.1 Data collection

The `DataCollection` class contains a collection of elements inheriting from `AData`. `AData` has a field of type `TimeStamp` that holds the timestamp when the piece of data was created. Thus `DataCollection` maintains a sorted collection of data and provides methods for adding and iterating through the collection. The `DataCollection` class is for example used for storing sensor samples and other data associated with a timestamp. See class `DataCollection` in [16] for more information.

3.5.2 Serialization

All data structures (`DataCollection` and `AData`) inherits from `ASerializable`, meaning it is possible to serialize and deserialize the object to and from a stream, this makes it possible to send them via the data link and to output and read them with standard stream operators in `C++`.

3.5.3 Sensor data logs

All sensor data classes inherit from the `DataCollection` class and contains private variables storing the sensor data values. The sensor data are serialized when they are logged to a file where the format of each sensor sample is of the form `timestamp data1 data2 ... dataN` and each sample is logged on a separate line. A description of each sensor data can be found below. Note that the laser sensor sample has a variable size as one sample contains points from one revolution which can be 0 to 360 points depending on the surroundings. In the following tables (Table 3.2 - 3.5) the serialization of each sensor data is shown.

All sensor data contain timestamps of when the data is recorded. This timestamp is divided into two parts, one displaying seconds and another displaying nanoseconds.

Table 3.2: Serialized GPS data

Column	Variable	Unit	Description
1	timestamp second part	s	The part of the timestamp displaying seconds
2	timestamp nanosecond part	ns	The part of the timestamp displaying nanoseconds
3	nmea	NMEA string	The raw NMEA data (GPGGA), containing position and accuracy data
4-6	RT90	RT90 coordinate	The RT90 position's x,y and z coordinates

The NMEA information is the GPS data in the NMEA format, which is a standard format for transferring GPS data.

Table 3.3: Serialized IMU data

Column	Variable	Unit	Description
1	timestamp second part	s	The part of the timestamp displaying seconds
2	timestamp nanosecond part	ns	The part of the timestamp displaying nanoseconds
3-5	rAcceleration	m/s^2	Raw acceleration reading
6-9	rAngularVelocity	rad/s	Raw Angular velocity reading, x,y and z coordinates
10-12	rMagnetic	Gauss	Raw magnetic field reading, x,y and z coordinates
13-15	sAcceleration	m/s^2	Stabilized acceleration data, x,y and z coordinates
16-18	sAngularVelocity	rad/s	Stabilized angular velocity data, x,y and z coordinates
19-21	sMagnetic	Gauss	Stabilized magnetic field data, x,y and z coordinates
22	roll	rad	roll angle
23	yaw	rad	yaw angle
24	pitch	rad	pitch angle

Table 3.4: Serialized Odometer data

Column	Variable	Unit	Description
1	timestamp second part	s	The part of the timestamp displaying seconds
2	timestamp nanosecond part	ns	The part of the timestamp displaying nanoseconds
3	delta_distance_left	m	Distance on left track since last reading
4	delta_distance_right	m	Distance on right track since last reading
5	velocity_left	m/s	Velocity of left track
6	velocity_right	m/s	Velocity of right track

Table 3.5: Serialized Laser data

Column	Variable	Unit	Description
1	timestamp second part	s	The part of the timestamp displaying seconds
2	timestamp nanosecond part	ns	The part of the timestamp displaying nanoseconds
3	Point1 distance	m	Distance to object
4	Point1 angle	degree	Angle to object
5	Point1 quality	-	Quality of measurement
...	...	...	...
k	PointN distance	m	Distance to object
k+1	PointN angle	degree	Angle to object
k+2	PointN quality	-	Quality of measurement

3.6 Operating system

Two versions of Ubuntu are used as operating systems in this project. The versions of Ubuntu that are used are Ubuntu Server 14.04 on Balrog and Ubuntu Desktop 14.04 on the base station.

3.6.1 USB serial devices

To enable ARM-processor communication, a file with the line `"options usbserial vendor=0x03EB product=0x6125"` have to be added to `/etc/modprobe.d/` [17]. The laser sensor works when plugged in and does not require anything special to enable communication.

To enumerate the same USB serial devices to the same devices in `/dev/` a UDEV rule have been used [18]. To set the speed of a specific serial device a similar rule have been used [19]. To be able to access the serial adapters the user has to be in the dialup group of the OS. The paths are shown in Table 3.6.

To configure all of the above run `installConfigs.sh` in the `os` folder on a newly installed Ubuntu machine.

Table 3.6: Device paths

Device	Path
IMU	`/dev/ttyBalrogImu`
ARM processor	`/dev/ttyBalrogArm`
GPS	`/dev/ttyBalrogGps`
Ultrasonic sensors	`/dev/ttyBalrogI2c`
Laser sensor	`/dev/ttyBalrogLaser`
Xbox button inputs	`/dev/input/js0`
Xbox rumble	`/dev/input/event3`

3.7 Map structure

This section describes the internal map structure used in the project. The map structure is used by Balrog and the base station alike.

The map structure consists of two separate maps: the probability map (a `ProbabilityMap` class type) and the obstacle map (an `ObstacleMap` class type). The maps functionality is in two different classes, `ObstacleMap` and `ProbabilityMap`, that inherits from the parent class `Map`. Since no object is of class `Map`, this class holds common functions without mutex lock or unlock for `ObstacleMap` and `ProbabilityMap`. The mutex lock down is done in the obstacle and probability map instead, to prevent more complex functions to get stuck in a mutex deadlock.

The purpose of the probability map is to keep track of the probability that Balrog has visited a specific part of the map while the obstacle map keep track of the lines found by SLAM, and which part of the map that holds obstacles.

Both maps are represented by a discrete grid, a 2D C-array. The grid size is decided when the map object is constructed. When constructing the object three arguments are given: width of the map (in full meters), height of the map (in full meters) and resolution (in number of grids per meter). If you create a map with the arguments 3, 4 and 2 the resulting grid map is 6 grids wide and 8 grids high, each grid a square with side length 0.5 m. A typical search area has the parameters of about (10,10,1) or (10,10,2). The obstacle map and the probability map uses the same basic map functionality, but there are some specific member functions for the different classes (see classes `ObstacleMap` and `ProbabilityMap` in [16] for further information).

We have not investigated the minimum grid width, but tests have been run with grids as small as 0.25 m wide, without resulting in any malfunctions. The `addLine()` function might get stuck in an infinite loop if the grid is to small and the line to be drawn is to long. This is due to numerical problems when operating on floats when calculating which grid positions are part of the obstacle and the free space between Balrog and the obstacle. See doxygen documentation [16] for `ObstacleMap::addLine()`.

Internal representation of the probability in the probability map is an `unsigned int`, but the external representation is a `double` between 0 and 1. In the obstacle map the status in each grid is represented by the statuses shown in Table 3.7

The obstacle map receives obstacle line position (in local coordinates) from SLAM and calculates which grids to be filled. It also fills the grid positions between Balrog and the obstacle as `EMPTY_TILE`. The probability of those tiles are not changed, so the route

Table 3.7: The possible statuses in the obstacle map.

Status	Numerical value	Description
UNEXPLORED_TILE	0	The initiated status of all tiles, if a tile is unexplored this indicates it
EMPTY_TILE	1	The status if a grid position is regarded as obstacle free
OBSTACLE_TILE	2	The status if a grid position contains an obstacle

planner will still evaluate the probability of those tiles and revisit if the probability is to low. When the route planner regards an object as closed all the grid position inside the obstacle is marked as OBSTACLE_TILE. The fill uses the flood fill algorithm with 4-way connectivity.

The mine positions are not represented by a map structure. The position for the mines that are found is saved in a vector that is drawn in the GUI. Balrog and the route planner does not take the mines in account at the moment, they are considered as EMPTY_TILE.

4 Base station

The base station consists of an ordinary computer running Ubuntu 14.04 which communicates with Balrog wireless over WiFi or via Ethernet cable. The main tasks for the base station are receiving and displaying data from Balrog in the GUI as well as sending commands to Balrog through a set of commands. The layout of the GUI is shown in Figure 4.1

The full user interface has been kept from last years project, but only a subset of the functionality has been ported to work with the new Balrog:

- It is possible to connect to Balrog by choosing File→connect and entering the IP address of Balrog
- It is possible to disconnect from Balrog by choosing File→disconnect
- It is possible to enter manual mode by clicking the button ”Start manual”
- It is possible to steer Balrog by the arrow keys in manual mode
- It is possible to increase the speed of Balrog with the vertical slider (or by pressing W) in manual mode
- It is possible to decrease the speed of Balrog with the vertical slider (or by pressing S) in manual mode

The rest of the buttons and components does not work at the moment since they are not integrated with the new Balrog software.

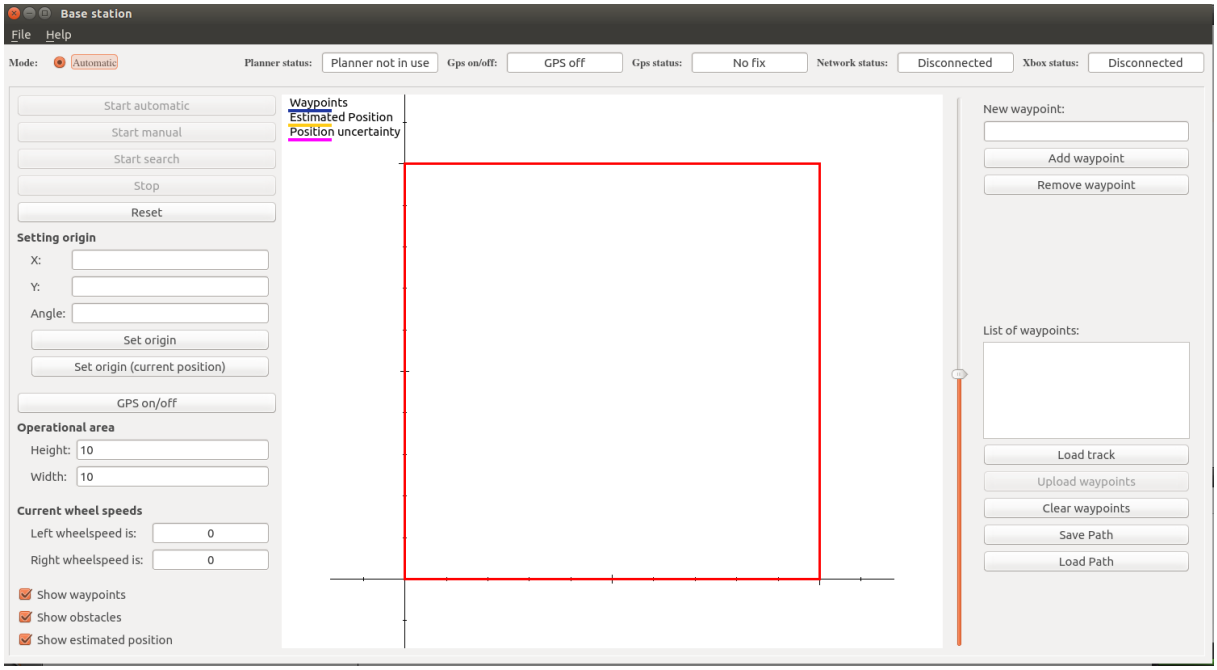

Figure 4.1: The graphical user interface

4.1 Communication

All data sent and received is encapsulated as shown in Table 4.1. There are messages with no payload, such as manual mode, and there are messages with payload, such as all sensor data messages were the payload is a serialized version if the sensor data object. Note that the payload can't exceed a length of $2^{16} = 65536$ characters.

Table 4.1: Data format for a message

ID	**Payload length**	**Payload**
uint16_t id	uint16_t len	char payload[len]

To receive a data message there are functions in the `NetCom` class which peaks the TCP sockets buffer to see if there are a whole message to be received and parsed, and a function that receives the data and calls the registered decoder of that particular ID. There is also a legacy implementation in which the id and stream are sent if no id and decoder and/or encoder pair is registered.

4.1.1 Balrog

On Balrog there is a system where one can register encoders and decoders. Using the `sendDataMessage` method in the `Communications` class the encoder of the data message is called and a data message is sent. On Balrog there is a communication thread which peaks and parses messages and after that sends States, Maps and Mines in a fixed tick.

4.1.2 Base station

On the base station the implementation is a bit different although using the same parent class(`NetCom`). The parsing is done using the legacy implementation, which is due to the use of signals. It would not be possible to register a signal if the instantiated objects were not known. The parsing is done aperiodicly using a socket to signal bridge. The signal is activated as long as there is data to read in the receive buffer of the socket.

4.1.3 Messages

In Table 4.2 and 4.3 is a summary of the data sent from Balrog to base station and base station to Balrog respectively.

Table 4.2: Messages sent from Balrog to base station

ID	Description
NETWORK_MESSAGE_STATE_DATA	Sends the latest state periodically
NETWORK_MESSAGE_WAYPOINTS	Sends the complete waypoints data structure each time it is updated
NETWORK_MESSAGE_OBSTACLE_MAP	Sends the complete obstacle each time it is updated (Note that this is not working correctly due to a bug in the serialization of the map)
NETWORK_MESSAGE_WHEEL_SPEED	Sends the current wheelspeed periodically
NETWORK_MESSAGE_MINE_POSITION	Sends the complete mines data structure each time it is updated

Table 4.3: Messages sent from base station to Balrog

ID	Description
NETWORK_MESSAGE_WAYPOINTS	Sends the complete waypoints data structure
NETWORK_MESSAGE_WHEEL_SPEED	Sends the current wheelspeed when it is changed
NETWORK_MESSAGE_SET_MODE_AUTOMATIC	Send a message indicating that autmatic mode is active
NETWORK_MESSAGE_SET_MODE_MANUAL	Sends a message indicating that manual mode is active

5 Hand controller

The hand controller facilitates the control of Balrog when in manual mode. The hand controller is a Microsoft Wireless Xbox 360 Controller which communicates at the 2.4 GHz band through a USB dongle located on Balrog.

5.1 Functionality

The main functionality of the hand controller is to be able to steer Balrog. This is done by using the left joystick on the controller. This can be done as long as the automatic mode has not been started. Several other functions that can be implemented are suggested in Section 11.

5.2 Implementation of communication

The communication between Balrog and the hand controller is built upon available Ubuntu drivers in combination with the C libraries `linux/input.h` and `linux/joystick.h`. The classes used for the hand controller communication are mainly `HandController` in the code for Balrog and `XboxButtonMap` in the common code. The same base is used as for the rest of the system.

When the hand controller connects to Balrog a file is created in the main system on Balrog. The file created is `/dev/input/jsX` where `X` is a number assigned to the hand controller. The file is interpreted with the C libraries mentioned above and from which the data can be translated into different commands.

HandController The `HandController` class handles the communication between hand controller and Balrog. This communication is handled in a separate thread and has a low priority, not to disturb other communication lines. The thread listens for communication from the hand controller by iterating a loop. The first part of this loop is reading the joystick file, which will block the thread until new data is available. When new data occurs the thread awakens and the command is interpreted. The information is then sent through via the shared data structures.

XboxbButtonMap The `XboxButtonMap` class handles the current configuration of the hand controller as well as communication regarding button configuration changes.

6 Balrog

This section focuses on the basic aspects of the crawler subsystem Balrog. It describes propulsion, the sensors available and the choice of operating system used on the main processing unit.

6.1 Conceptual overview and definition of terms

This section presents an overview of the implementation of the software running on Balrog.

The key concepts for the implementation is summarised in the following list:

- Each subsystem must be independent of the other subsystems and must be able to run as a standalone application. All dependencies must be passed in to the constructor at instantiation. This allows a high degree of decoupling between the different subsystems and they can be developed in parallel and tested independently of each other.
- A subsystem is run in its own thread and can either run periodically at a specified frequency or once when they are called.
- Shared data structures can be shared between subsystems by passing in their reference. All operations in the shared data structures are implemented as if they were atomic to avoid concurrency problems and race conditions.

Figure 6.1 shows an overview of the different parts and how they interact.

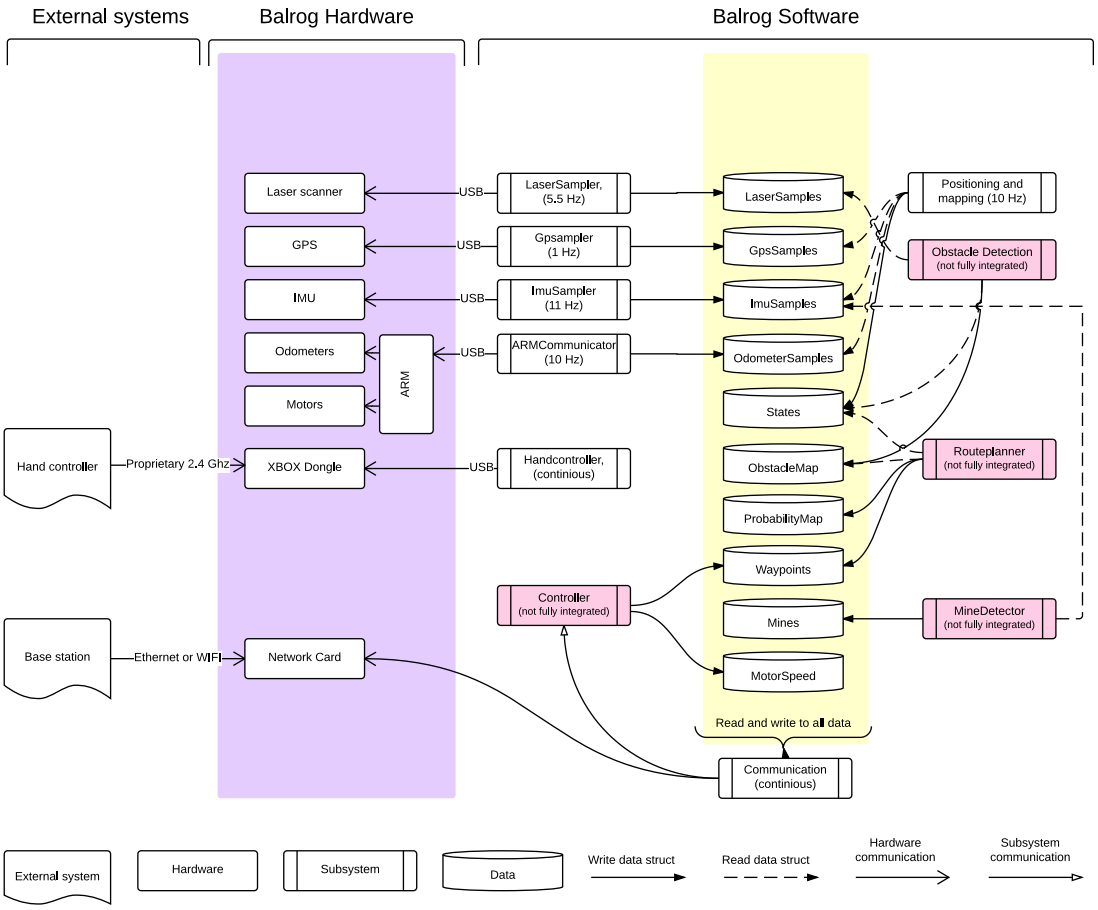

Figure 6.1: Illustration of subsystems, shared data structures and hardware. An arrow pointing from a subsystem to a data structure indicates that the subsystem is writing to the data structure. An arrow pointing from a data structure to a subsystem indicates that the subsystem is reading from a data structure. A red box indicates that the subsystem is not fully integrated.

6.2 Computer

Balrog is equipped with a computer with the following specifications:

- Processor: Intel Core i7 3.4 GHz, quadcore with hyper threading
- Memory: 8 GB DDR3 RAM
- Storage: 250 GB SSD hard drive of which 80GB is used for the Ubuntu partition
- Operating System: Ubuntu 14.04

6.3 Subsystems

Subsystems inherits from the class `ARunnable` and are all completely isolated from each other. As long as they are instantiated with the correct parameters (such as references to shared data structures and other subsystems), they must be able to run independently of the other subsystem. This allows a high degree of decoupling between subsystems and they can easily be tested and developed in parallel.

Each subsystem runs in its own thread. The thread instantiation is taken care of automatically in the `ARunnable` class and does not need to be considered when using the objects. Each subsystem may in turn spawn new internal subsystems based on the same principle as above.

Subsystems may need references to shared data structures in order to read or manipulate them. In addition, subsystems may also need references to other subsystems. For example the `Communication` subsystem carries a reference to the controller, making it possible to pause and resume the controller on command from the user. See the doxygen documentation for each class [16] for further information.

6.4 Main entry point Balrog

The execution of the Balrog starts in the main method in the `app_balrog` project. The main method takes responsibility for some global initialization, creation of shared data structures as well as creating, initializing and starting the high level subsystems. The subsystems themselves may spawn their own subsystems during execution.

A brief overview of the steps taken in this file can be found in Figure 6.2.

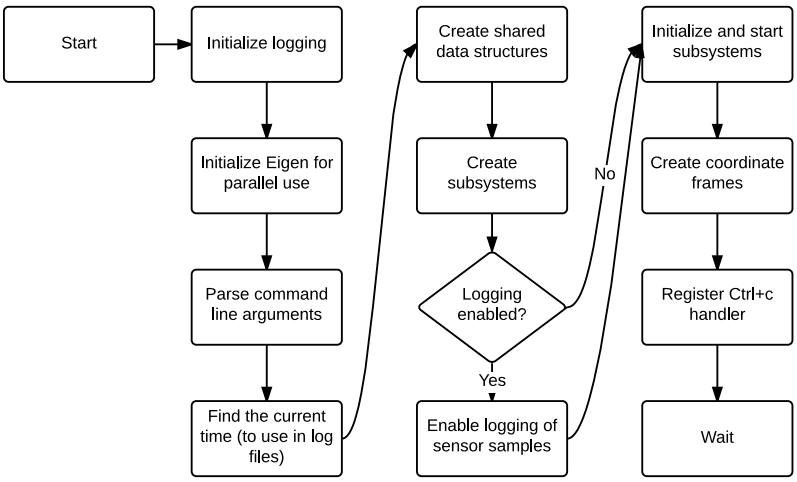

Figure 6.2: Brief overview of main method

6.5 Propulsion

Balrog is able to navigate by having two tracks equipped with an electrical motor for each track. There are also two odometers, one for each track, recording the rotation of the tracks. The communication and control with the motor and odometers is facilitated via an ARM processor connected to the main control unit via an I2C bus [20].

6.5.1 ARM-processor

An ARM processor is responsible for reading odometer data and setting/retrieving the current motor output. The code in this ARM processor was not accessible to us and we therefore needed to use the external interface already provided. This external interface is very poorly documented from previous projects but it is clear that the ARM is connected to the main unit via BL233B [21] I2C to PC converter and that a text based protocol with certain commands [20] is used to communicate with the ARM processor.

6.5.2 ARM-processor commands

The communication protocol is text based. In Table 6.1 the different communication commands for the ARM-processor are displayed.

Table 6.1: Communication commands for the ARM-processor

command	format	parameters	return
Set velocity	`sVXXYY`	XX and YY is velocity in mm/s (one byte each) for left track and right track respectively. Last year's project only used speeds in the range of 100 mm/s and 760 mm/s.	sVOK if successful
Get distance	`upDi`	Retrieve distance traveled since last read.	Signed double for each track in mm.*
Get velocity	`upVe`	Retrieve current velocity	Signed double for each track in mm/s.*

* The lack of comments in last year's code inflicts uncertainty in the interpretation of the format of the data retrieved from the odometers, using `upDi` and `upVe`. In `odometers.cpp` (from 2013 years implementation) the raw data from the odometers is divided by the factor 10 000. This finding together with the fact that the precision of the mark spacing is given in $\frac{1}{10000}$ mm makes it probable that the format of the data returned by `upDi` and `upVe` is mm and mm/s respectively.

6.6 Sensors

The following subsection describes the different sensors mounted on Balrog.

6.6.1 GPS

Balrog has a GPS receiver of the type Ublox-5 [22]. The receiver sends data through a serial port according to the NMEA-0183 standard which is a text based protocol developed for navigation equipment. The different NMEA messages used are available on pages 40-54 in reference document [22]. The information sent contains position data, time of positioning, number of satellites used for positioning and positioning accuracy.

The positioning data is given as latitude and longitude according to the WGS84 standard. These coordinates are converted to the RT90 standard, using the algorithms from last year, before being used for SLAM, since this is the standard used throughout Balrog [20].

6.6.2 Laser sensor

Balrog is equipped with a laser sensor from Robot Peak, RPLIDAR A1M1-R1, displayed in Figure 6.3. The laser sensor rotates 360° in positive θ direction as is shown in Figure 6.5. It samples distance data from obstacles at a distance of up to 6 m. The revolution frequency of the sensor is adjustable from 1 Hz to 10 Hz. The maximum frequency for which the sensor collects data samples for every degree (360 samples per revolution) is 5.5 Hz[23].

Figure 6.3: Laser sensor mounted on Balrog.

Connections

The connection between the laser sensor and the main control unit is over a USB cable through a USB converter (Silicon Labs CP2102) [24]. The USB connection is configured as a virtual COM port on the main control unit. The USB cable also provides the laser sensor (5 V) and its motor (3.6 V - 6 V) with power.

Consumption

In Table 6.2, the voltage and current used by the laser sensor is specified [23].

Table 6.2: Power supply and consumption of the laser sensor.

Item	Unit	Min	Typical	Max	Comments
Scanner system voltage	V	3.6	5	6	Low ripple voltage recommended
Scanner system current	mA	TBD TBD	40 130	70 200	Sleep mode (5 V input) Work mode (5 V input)
Motor system voltage	V	3.6	5	6	Adjust voltage according to speed
Motor system current	mA	TBD	100	TBD	5 V input

Messages

The laser sensor has three different message protocols: no-reply, single-reply and multiple-reply. The no-reply requests are `STOP` and `RESET`. The single-reply requests are `GET_INFO` and `GET_HEALTH`. The multiple-reply requests are `START_SCAN` and `FORCE_SCAN`. See Table 6.3 for details [25].

Table 6.3: Message requests to the laser sensor.

Request name	Value	Response mode	Description
`STOP`	0x25	No	Stops the laser sensor.
`RESET`	0x40	No	Resets the laser sensor core.
`GET_INFO`	0x50	Single	Ask for laser sensor information, (e.g. serial number, hardware version).
`GET_HEALTH`	0x51	Single	Ask for laser sensors health info.
`START_SCAN`	0x20	Multiple	Starts scan by entering the scanning stage.
`FORCE_SCAN`	0x21	Multiple	Starts scan without checking rotation speed.

Returned data

When in scanning mode, the laser sensor returns data after each completed revolution. The data returned is 40 bits for each sample value. The data contains 6 bits for the quality, 15 bits for the angle, 16 bits for the distance, 2 bits for start new scan and 1 control bit. The quality bits represents the strength of the returned laser pulse. The angle (θ) is given in degrees from the forward position and the distance (d) to an object is given in millimetres. In Figure 6.5 the returned angle and distance is displayed. The start bit S is sent with a complementary inverted $\bar{S}$ and the control bit, C, is always 1. See Figure 6.4 for the bit order. For return data for the other requests (`GET_INFO` and `GET_HEALTH`), see *RPLIDAR Interface Protocol* [25]

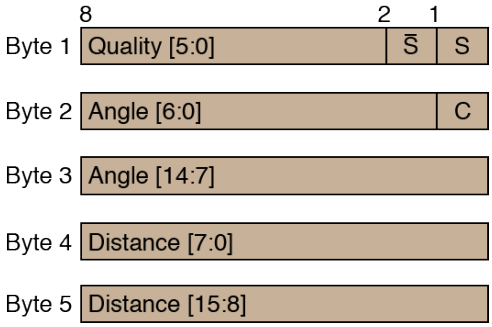

Figure 6.4: Returned data from laser sensor for each sample.

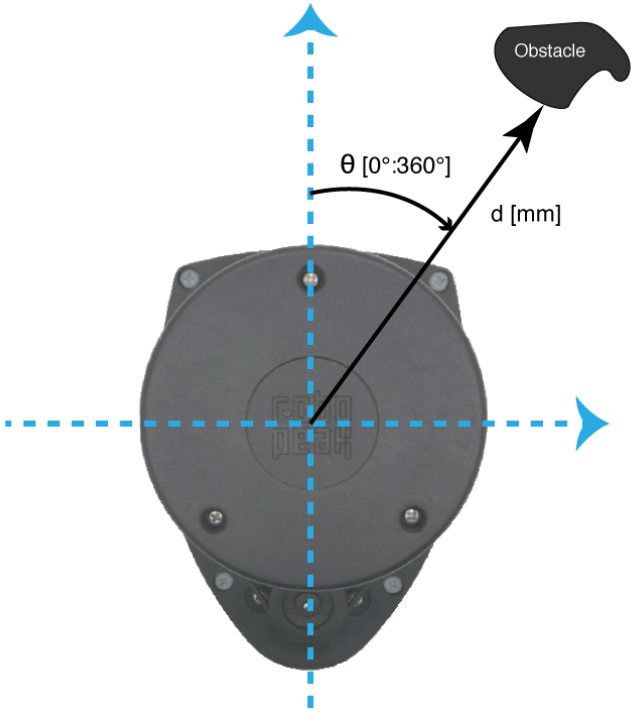

Figure 6.5: Graphic display of measured angle and distance by the laser sensor.

Assembly

The laser sensor is mounted above the computer with a safety cover. The cover consist of four legs/poles and a roof. See Figure 2.1 for an image of the final assembly.

Implementation

The laser sensor API uses RoboPeaks development kit source code as library for the functionality. The library uses a `driver` instance, an object that is created and is operated upon to communicate with the hardware. The driver, after it is connected, uses four main functions: `getHealth()`, `startScan()`, `grabScanData()` and `ascendScanData()`.

- `getHealth()` - Check laser sensors health and returns a health code (0 if ok)
- `startScan()` - Starts the background RPlidar thread for sampling the sensor

- `grabScanData()` - Check if laser sensor has new data, returns RP code `RESULT_OK` if new data is available
- `ascendScanData()` - Saves the scan data in an array

The function `work()` in the class `LaserSampler` samples the laser sensor and saves the sample in a `LaserDataPoint`-object. If the quality of the sample is zero, the sample is discarded and not saved in a `LaserDataPoint`. A `LaserDataPoint` is a simple class that contains fields for distance $[m]$, angle $[^\circ]$ and quality $[-]$. All `LaserDataPoints` from one revolution is then stored in a `LaserSample`

Noise model

To make a correct decision based on the information given by the laser sensor, a model of the noise was developed. This model is to be used in the state space model used for SLAM to improve the modelled behaviour. Result and further information is found under the SLAM section, 7.2.3.

6.6.3 IMU

Balrog is equipped with a 3DM-GX1 IMU [26] manufactured by MicroStrain.

The computer on Balrog communicates with the IMU through a USB-connection and can retrieve the following data from the IMU:

- Raw acceleration data along all three axis (± 5 m/s^2)
- Raw magnetic field data along all three axis (± 1.2 Gauss)
- Raw angular velocity data around all three axis ($\pm 300^\circ/s$)
- Stabilized and preprocessed acceleration, magnetic and angular velocity data, along with estimation of absolute angle

The resolution of each measurement is 16 bits.

According to the previous year, the acceleration data is noisy and biased, and affected by roll and pitch. The noise is compensated for in the state vector $\mathbf{x}$ (see 7.4), where the bias is modulated, and the effect of gravitation has to be taken into account. The IMU is also able to give a pre-filtered signal which is less noisy, and could be regarded as an output option.

The angular velocity of Balrog is estimated using the gyro in the IMU. As the angular velocity is biased with a non-specific mean (see [20, sec 6.1.3], bias from the output is modelled.

The magnetic field strength measured by the magnetometer of the IMU is a sum of the magnetic field of the earth, external fields from the mines and disturbances generated by the electric motors. Subterranean mines produce clear spikes in the measured field, thus the magnetic field data can be used for mine detection. Unfortunately, the motor disturbances are in the same magnitude as the magnetic field of the earth, which makes it hard to use the magnetometer for the purpose of navigation.

A specification of the performance of the MicroStrain 3DM-GX1 IMU can be found in Table 6.4.

Table 6.4: Performance of the MicroStrain 3DM-GX1 IMU

Parameter	**Value**
Non-linearity accelerometer	0.2 %
Bias stability accelerometer	0.010 g
Non-linearity gyro	0.2 %
Bias stability gyro	$0.7\,^\circ/s$
Non-linearity magnetometer	0.4 %
Bias stability magnetometer	0.010 Gauss
Resolution orientation	$< 0.1^\circ$ minimum
Repeatability	$0.20\,^\circ$
Accuracy	$\pm 0.5\,^\circ$ at static test conditions
	$\pm 2.0\,^\circ$ at dynamic (periodic) test conditions

A detailed technical specification is available online [27] as well as in the communication manual [28].

The IMU is suffering from spikey measurements for undisclosed reasons as shown in Figure 6.6.

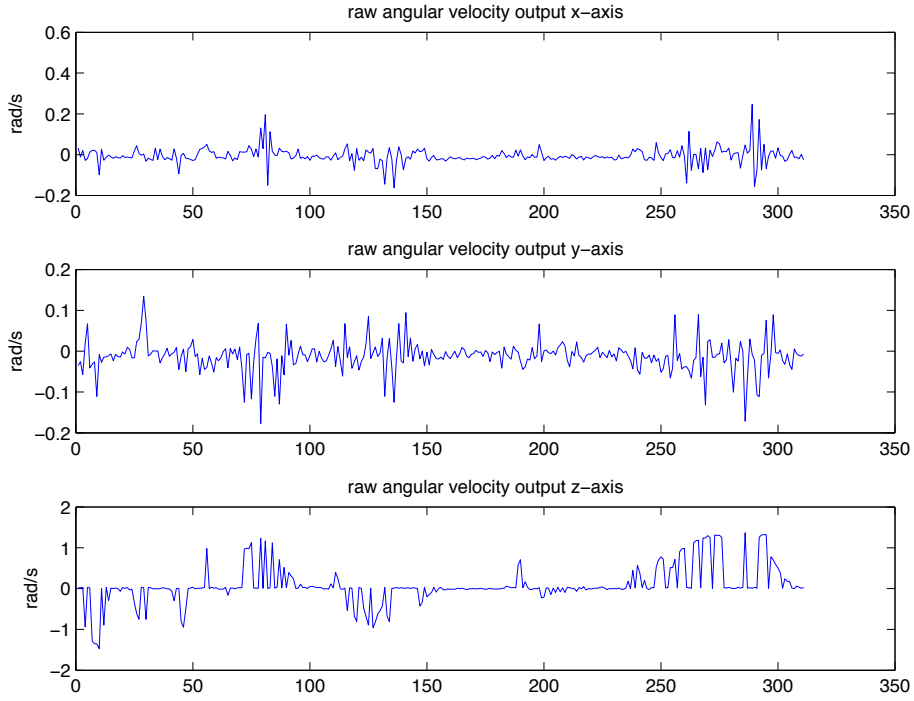

Figure 6.6: Registered angular measurements from the IMU

This has to be fixed. Using this data won't improve the state estimates of the balrog and will probably just make it worse.

6.6.4 Odometers

The odometers measure the distances travelled by each track if there were no slip. Each odometer has 500 marks spaced with 1.0744 mm that are used to estimate the distance travelled [20].

7 SLAM

The SLAM subsystem is responsible for estimating the position of the robot, mapping the environment, identifying objects and associate them with landmarks.

The subsystem is further divided into three additional systems explained below: Positioning, Mapping, Obstacle detection.

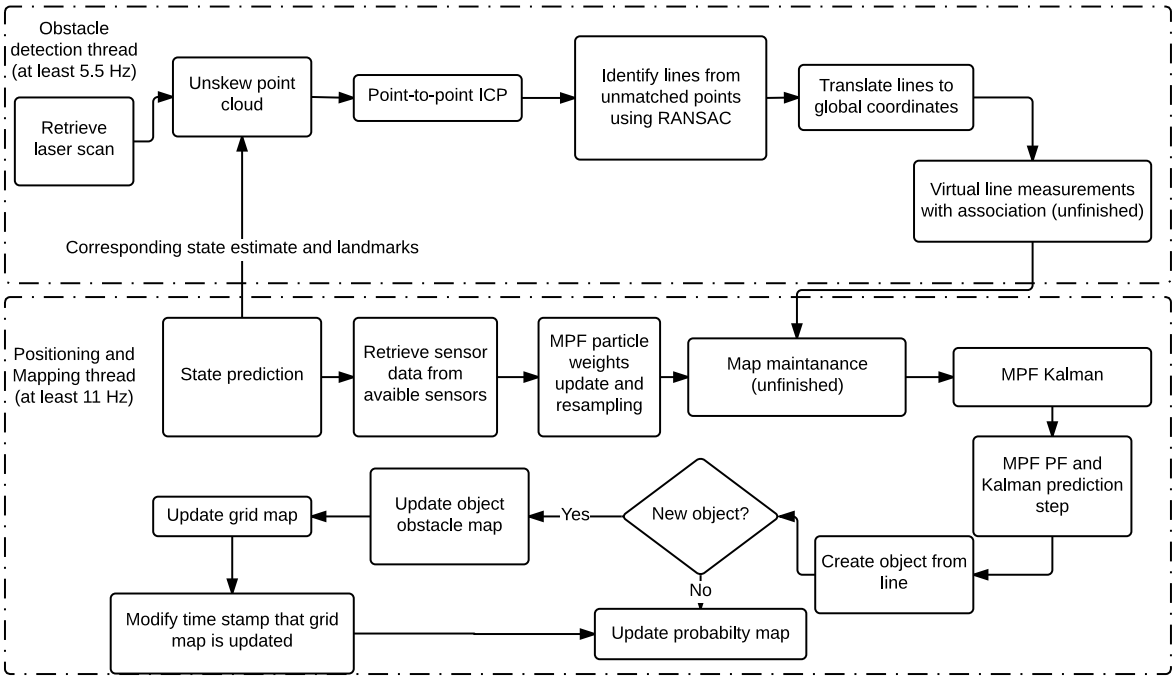

Figure 7.1: Flowchart of SLAM parts: Obstacle Detection, Positioning and Mapping.

7.1 Obstacle detection

The obstacle detection subsystem is responsible for taking raw measurement values from the laser sensor and pre-processing them. The processed measurements are then inputs to the positioning and mapping subsystems.

Pre-processing of the laser scanner includes removing outliers, compensating for measuring while moving and approximating the point cloud with straight lines representing objects. The compensation for vehicle movement while scanning (unskewing) is done by calculating an appropriate roto-translation based on the velocity, heading and heading angular velocity of the vehicle kept during the full scan.

Removal of outliers is done firstly by checking which laser scan points that are too close to close to the sensor (< 20 cm), which indicates that they are in fact part of the tracked vehicle. These are removed, along with any points that have their closest distance at a distance greater than 10 cm. The purpose of this is to avoid disturbances from phantom points, created by noise or small objects outside, e.g. a tall grass straw.

The point cloud created by the laser sensors consists of 360 angle values and a distance associated to each of them. Based upon this, a probabilistic sensor model, that connects

each distance with the probability of it being a correct measurement is calculated, will be constructed. The noise model is described in Section 6.6.2 on page 22.

7.1.1 Line extraction

In order to separate the different lines to be created, the edges of each line must be identified. To solve this, the following algorithm , based upon [29, algorithm 4 p.3] has been implemented:

1. Start iterating through the all available measurements in one laser sample (one laser scan rotation)
2. When a distance measurement over the threshold (currently 7.5 cm) is found, mark this point as an edge of a line. This is a question of tuning, and it has been performing well, mainly in outdoor environments.
3. Continue the iteration, if the distance to the next point is greater than the threshold mark this point as the other edge of the line
4. Start searching for a non-maximum distance measurement again, and repeat 2. - 3. until the whole lap is completed
5. To be sure of detecting edges immediately in front of the scanner, complete the lap until you reach the firstly found edge again

This algorithm is further specified with a flow chart in Figure 7.2.

With this method, the line extraction algorithms will have the necessary information to separate the lines to be created.

RANSAC A RANSAC algorithm [30] has been implemented in order to extract a suitable line given a region of a laser scan, defined by its two edges.

The RANSAC (RANdom SAmple Consensus) algorithm can perform robust fitting in a situation where outliers are present. The laser sensor has a risk of producing outliers when it finds a false point. Given a set of data, it performs the following operations:

1. Randomly select a subset of two points of the data
2. Create a line through these two points
3. Calculate the distance between the remaining points and this line
4. Split the set in inliers and outliers using this distance
5. If the inlier set is sufficiently large, use it to fit a new line and remove the inliers
6. Repeat until max iterations reached or there are too few points left to work with

This algorithm is further specified with a flow chart in Figure 7.2.

In summary, it tries different combinations of subsets until a consensus is reached concerning which points are inliers and which are outliers by separating the points in different subsets and comparing them to each other.

The current RANSAC algorithm performs well for straight indoor obstacles (e.g. a shelf or a box standing on the floor). It has also performed well in detecting obstacle of curved shapes. However regarding extended walls consisting of more than 50 laser sample points, the algorithm will sometimes reach max number of iterations and produce a line that is not very representative of the object. This should not cause any problems, since extended walls are not really common in outdoors environments.

The lines are sent to the obstacle map in the form of coordinates for the start and ending position of the line, together with the current relative position of Balrog.

In Figure 7.3 a raw laser scan from the LIDAR can be studied. It is taken in an office

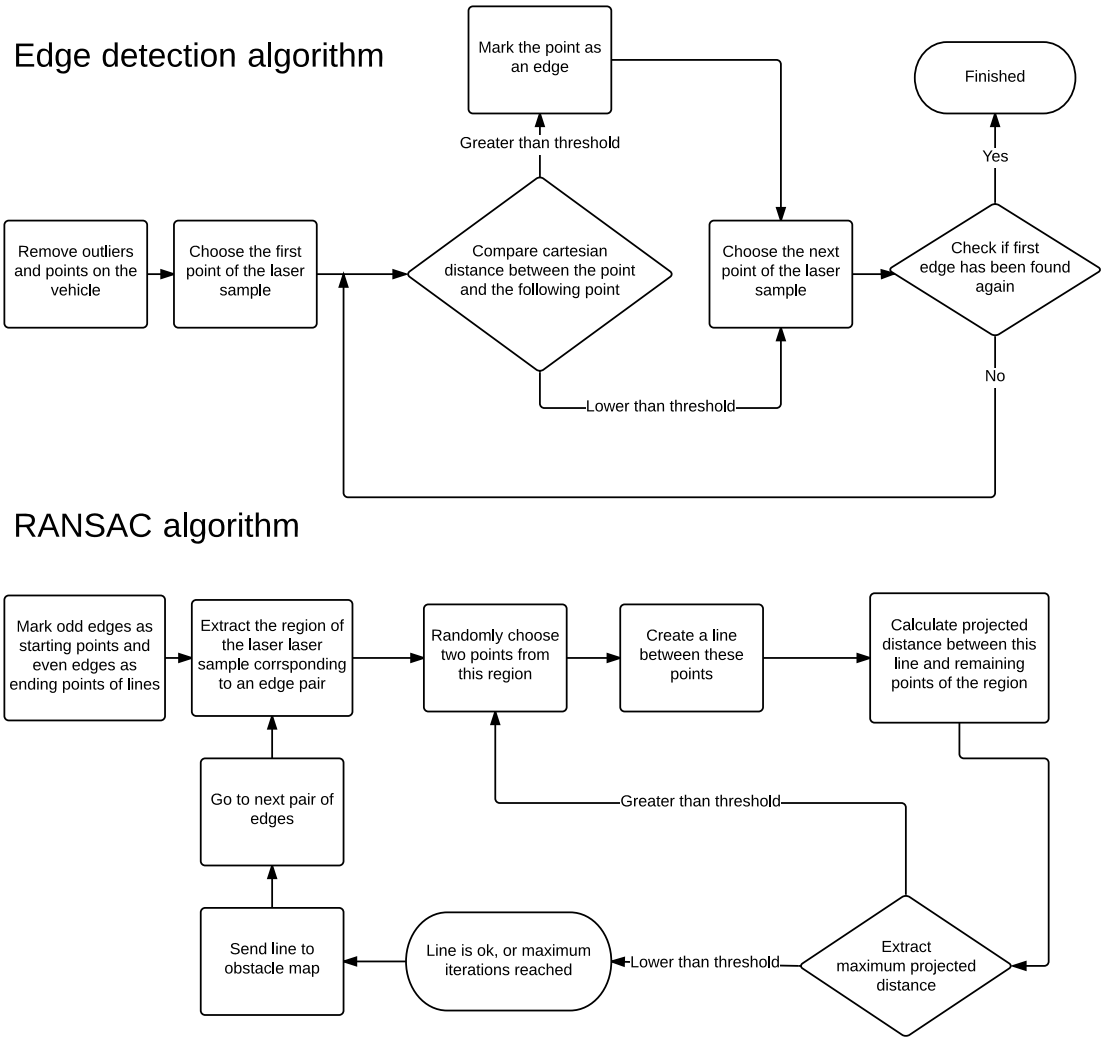

Figure 7.2: Flowchart of algorithms for edge detection and line extraction.

equipped with three obstacles (two shelves and a cardboard box). The noisy area around the left hand wall origins from it being equipped with radiators.

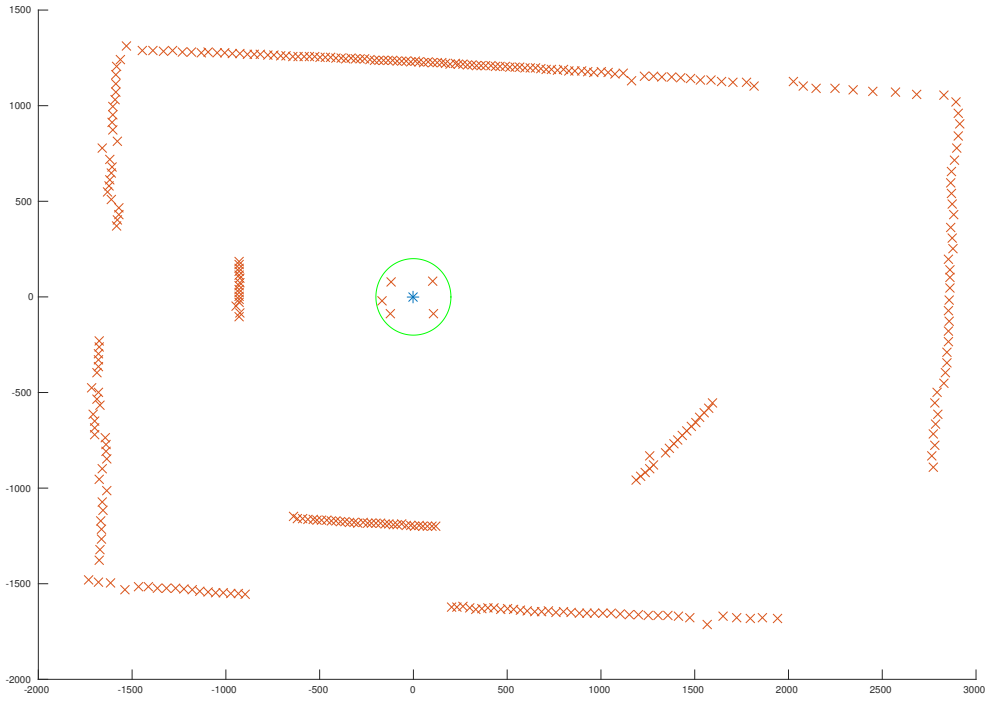

Figure 7.3: A raw laser scan in a room, measured length data represented by red crosses. Balrog is marked by a blue star and a green circle.

In Figure 7.4 edge detection and RANSAC have been performed, resulting in edges (diamonds) and lines (yellow lines). Two of the obstacles are represented by a single line, while the tilted object resulting in two separate lines. This is a question of tuning, defining when two objects are separate or not.

Theoretically there should be no problems regarding obstacles that are just a few centimetres from the Balrog. Distances closer than 40 cm have not been thoroughly tested, but there is no reason that they should not be detectable.

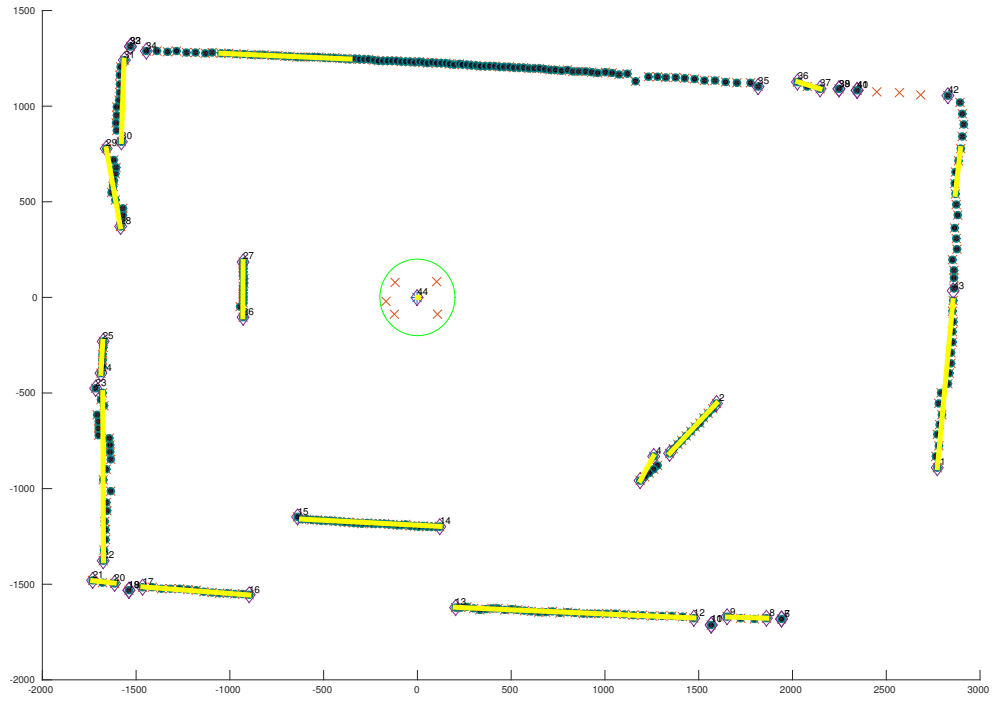

Figure 7.4: Lines extracted from the laser scan data in Figure 7.3, marked in yellow. Measured length represented by red crosses. Detected edges are marked by a purple diamond. Exluded points are marked by an orange ×. Balrog is marked by a blue star and a green circle.

7.1.2 Landmark model

The landmarks will be stored in a vector when identified. The objects can be regarded as a state and will therefore be modelled as

$$\mathbf{m}_{k+1} = \mathbf{m}_k, \tag{7.1}$$

where the landmark vector

$$\mathbf{m}_k = \begin{pmatrix} m^1 \\ m^2 \\ \vdots \\ m^j \end{pmatrix} \tag{7.2}$$

, with j being the number of known landmarks.

The landmarks will be stored as lines represented by two coordinates (x, y). Every line (when seen) will be used for positioning, so lines will not be fused together creating objects (polygons).

7.2 Positioning

The positioning subsystem is responsible for estimating the position, with regards to measurements from the IMU, GPS and odometer; output from the obstacle detection

subsystem; the map from the mapping subsystem and a motion model of Balrog.

The measurements from the IMU, GPS, odometer and ultrasonic sensor are updated with a frequency of 11 Hz and the laser sensor with a frequency of 5.5 Hz.

7.2.1 Motion model

The dynamics of the Balrog can generally be modelled as

$$\mathbf{x}_{k+1} = f(\mathbf{x_k}, \mathbf{u_k}, \mathbf{w_k}), \tag{7.3}$$

where the non-linear function f describes the dynamics of Balrog, with regards to the process noise $\mathbf{w_k}$, the input to the system $\mathbf{u_k}$ and the state vector $\mathbf{x_k}$.

As modelling the acceleration was more challenging than rewarding the decision became to implement a coordinated turn constant velocity with bias model as described in the next section because the values from the IMU are biased with a non-specific mean (according to [20] page 25-27).

Coordinated turn constant velocity with bias model The state matrix $\mathbf{x_k}$ is now

$$\mathbf{x_k} = \begin{pmatrix} x_k \\ y_k \\ \psi_k \\ v_k \\ \omega_k \\ \omega_k^b \end{pmatrix}, \tag{7.4}$$

where x_k and y_k is the two dimensional local position of the Balrog, ψ_k the heading, v_k the speed of the balrog, ω_k the turning rate of the balrog and ω_k^b the bias of the turning rate.

The dynamics then becomes

$$\mathbf{x_{k+1}} = \begin{pmatrix} x_k + Tv_k\cos(\psi_k) \\ y_k + Tv_k\sin(\psi_k) \\ \psi_k + T\omega_k \\ v_k \\ \omega_k \\ \omega_k^b \end{pmatrix} + \mathbf{w_k}, \tag{7.5}$$

with the process noise $\mathbf{w}$ (noise in v_k, ω_k, ω_k^b) affecting the states as

$$G = \begin{pmatrix} \frac{T^2}{2}\cos(\psi_k) & 0 & 0 \\ \frac{T^2}{2}\sin(\psi_k) & 0 & 0 \\ 0 & \frac{T^2}{2} & 0 \\ T & 0 & 0 \\ 0 & T & 0 \\ 0 & 0 & T \end{pmatrix}, \tag{7.6}$$

7.2.2 Measurement model

The measurements will be modeled as,

$$\mathbf{y_k} = h(x_k, \mathbf{m}_k) + e_k, \tag{7.7}$$

where $h(x_k, \mathbf{m}_k)$ is a non-linear function dependent on both the state vector $\mathbf{x}_k$ and the landmarks $\mathbf{m}_k$, and e_k is the measurement noise.

Modelled bias As we're not using any acceleration measurements but are modelling the bias, we get

$$\mathbf{y_k} = \begin{pmatrix} X_{\text{GPS}} \\ Y_{\text{GPS}} \\ \omega_{\text{IMU}} \\ \frac{v_r + v_l}{2} \\ \frac{v_r - v_l}{d_v} \\ m^1_{\text{rk}} \\ m^1_{\alpha} \\ \vdots \\ m^j_{\text{rk}} \\ m^j_{\alpha} \end{pmatrix}, \tag{7.8}$$

where X_{GPS} and Y_{GPS} are measured using the GPS values. These are updated at a frequency of 1 Hz. From the IMU we achieve a measurement of the turn rate of Balrog. From the odometers the speed of Balrog is measured, by taking the mean of left and right. Also, the turn rate can be approximated calculating $\frac{v_r - v_l}{d_v}$, where d_v is a virtual width, as per last year. Both the odometers and IMU is updated with a frequency of 10Hz. With the laser sensor the distance m^i_{rk} and angle m^i_{α} to each known landmark $i = 1, 2, \ldots, j$ is measured. The laser sensor is updating with a frequency of 5.5 Hz, which corresponds to half of the other sensors with exception of the GPS.

The measurement function $h(\mathbf{x_k}, \mathbf{m_k})$ is implemented as

$$h(\mathbf{x_k}, \mathbf{m}_k) = \begin{pmatrix} x_k \\ y_k \\ \omega_k + \omega^b_k \\ v_k \\ \omega_k \\ \sqrt{(m^1_x - x_k)^2 + (m^1_y - y_k)^2} \\ \text{atan2}(m^1_y - y_k, m^1_x - x_k) \\ \sqrt{(m^2_x - x_k)^2 + (m^2_y - y_k)^2} \\ \text{atan2}(m^2_y - y_k, m^2_x - x_k) \\ \vdots \\ \sqrt{(m^j_x - x_k)^2 + (m^j_y - y_k)^2} \\ \text{atan2}(m^j_y - y_k, m^j_x - x_k) \end{pmatrix}, \tag{7.9}$$

where atan2(y, x) corresponds to the function $\arg(x + iy)$ with $0 \leq \arg(x + iy) \leq 2\pi$. As the IMU gives biased values of the turning rate and the acceleration, these will be modelled in the measurement model. This is an extension of the measurement function with the landmarks (both distance and direction) being added to the function h.

The covariance of the measurement noise will vary whether we're standing still or not. If we are standing still, the speed measurement will rely heavily on the measurement of the odometers, and thus neglecting any measurements coming from the gyro. Vice versa if we're not standing still (the odometers are unreliable when turning because of slip).

7.2.3 Noise model

Since the laser sensor is new hardware in this project a noise model of will be developed to increase the performance of the filter used. The noise model will be approximated as Gaussian for the sake of filtering.

The variance was calculated by letting Balrog collect over 100 revolutions of data. The surroundings was a rectangle shaped room of approximately 4x5m with a book shelf in one of the corners. Mean length and mean standard deviance is displayed in Figure 7.5.

The model was chosen to focus on the length measurement noise and does not take into account the uncertainty of the angle measured. The noise in the laser sensor's length measurements are assumed to be white and Gaussian with a mean of zero.

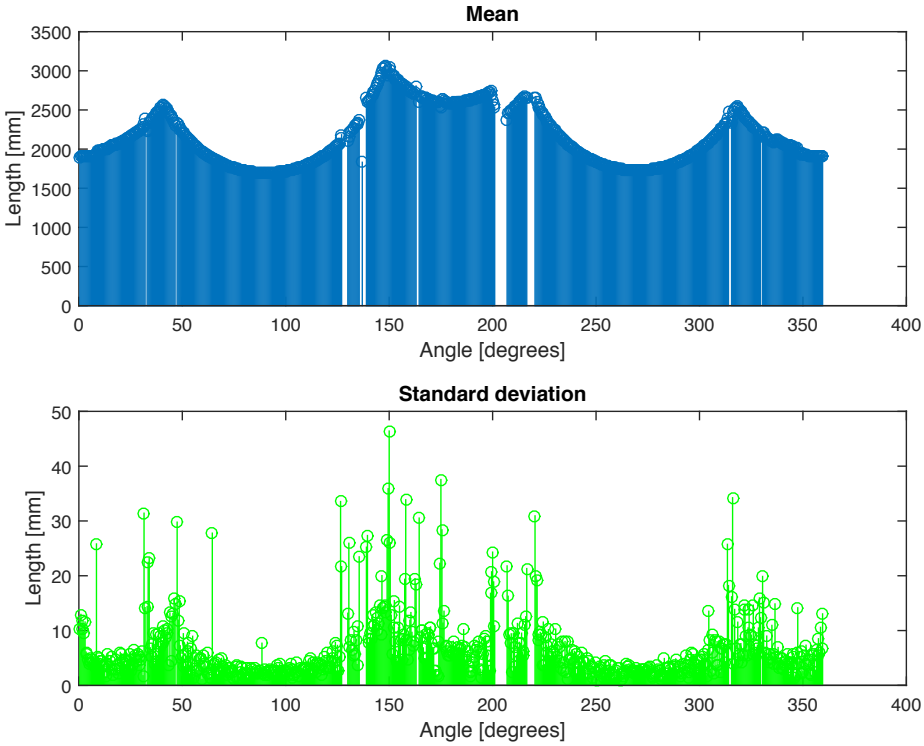

Figure 7.5: Mean length and mean standard deviation as function of measured angle.

Figure 7.5 displays a small standard deviance and a relation between standard deviation and length measured is visible. By the corners (at values of about 40, 150, 220, 320

degrees) the uncertainty of measured length is higher. The angle uncertainty causes the length to be measured in a small spectrum of true angles, an effect which in corners have a greater impact because of the sharp variation.

The relation between variance and measured length is further investigated in Figures 7.6 and 7.7 where a linear regression and quadratic regression has been applied respectively.

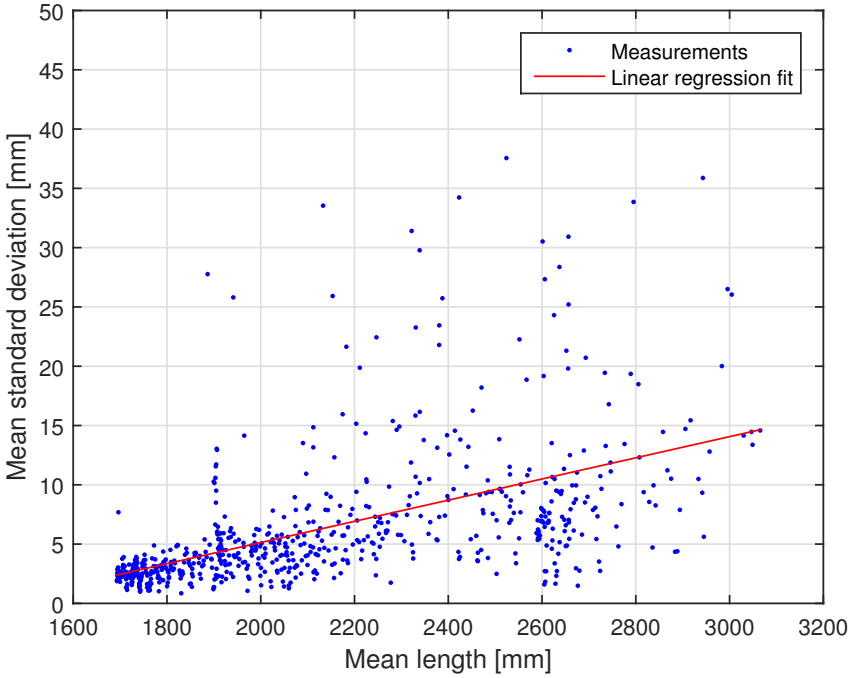

Figure 7.6: Linear regression of length/variance dependency.

The linear model in 7.6 is represented by the polynomial $f(x) = p_1 \cdot x + p_2$ and has the following parameters (with 95% confidence bounds):
$p_1 = 0.00893\ (0.007888, 0.009972)$
$p_2 = -12.73\ (-14.99, -10.46)$

The goodness of fit is
R-square: 0.2938
Adjusted R-square: 0.2928
RMSE: 4.939

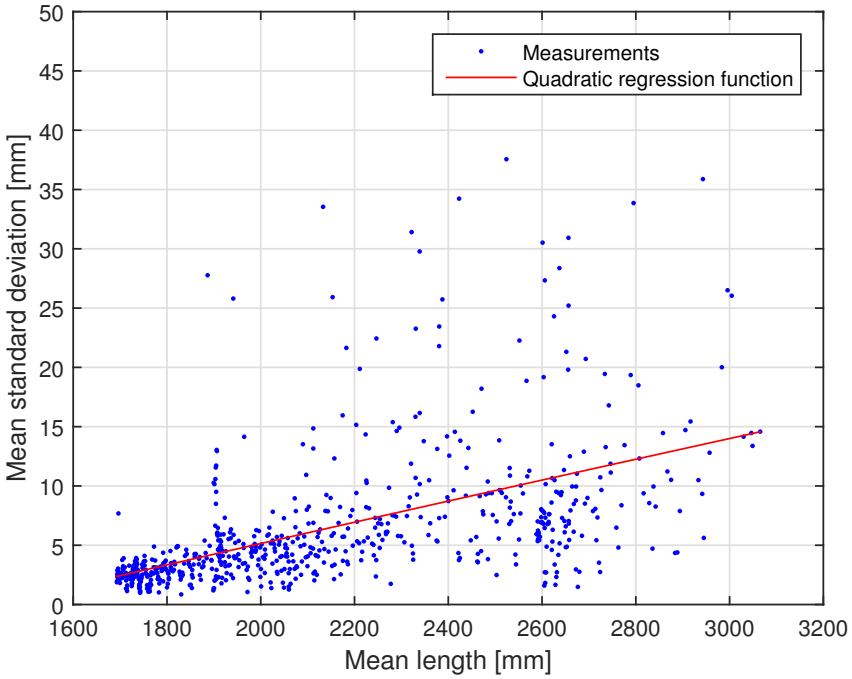

Figure 7.7: Quadratic regression of length/variance dependency.

The quadratic model in Figure 7.7 is instead represented by $f(x) = q_1 \cdot x^2 + q_2 \cdot x + q_3$ and has the following parameters (with 95% confidence bounds):
$q_1 = -1.531 \cdot 10^{-7}$ $(-3.35 \cdot 10^{-6}, 3.043 \cdot 10^{-6})$
$q_2 = 0.009618$ $(-0.004782, 0.02402)$
$q_3 = -13.48$ $(-29.33, 2.378)$

The goodness of fit is then
R-square: 0.2938
Adjusted R-square: 0.2917
RMSE: 4.943

As seen in Figures 7.6 and 7.7 there is no big difference between the two models when inspecting the graphs and their fit to data. In fact, the value of R-square is the same which indicates that the model in this case doesn't improve when adding complexity to it. When looking at the parameters it is clear that the parameter $q1$ does not contribute to the quadratic model since the value is close to zero and the confidence interval contains zero. The relation between the length measured and the standard deviation of measurement is then considered to be linear.

7.2.4 Filter

As there are both linearities and non-linearities in both the motion function f and measurement function h, a marginalized particle filter is implemented.

Marginalized particle filter To implement the filter, the states are augmented as

$$\mathbf{z}_k = \begin{pmatrix} (\mathbf{x}_k^n)^{\mathrm{T}} & (\mathbf{x}_k^l)^{\mathrm{T}} & \mathbf{m}_k^{\mathrm{T}} \end{pmatrix}^{\mathrm{T}} \tag{7.10}$$

where $\mathbf{x}_k^n$ is the non-linear part of the state vector $\mathbf{x}_k$, $\mathbf{x}_k^l$ the linear part and $\mathbf{m}_k$ the landmarks (which are linear(ised)). The linear Gaussian parts will then be estimated using a Kalman filter. See [31, eq. 7 p. 2] for a mathematical description.

Augmenting and spacing x_{k+1} in 7.5 and $\mathbf{m}_{k+1}$ in 7.1 as

$$\mathbf{z}_{k+1} = \begin{pmatrix} x_{k+1} \\ y_{k+1} \\ \psi_{k+1} \\ v_{k+1} \\ \omega_{k+1} \\ \omega_{k+1}^b \\ m_{k+1}^1 \\ \vdots \\ m_{k+1}^j \end{pmatrix} = \begin{pmatrix} x_k + Tv_k\cos(\psi_k) \\ y_k + Tv_k\sin(\psi_k) \\ \psi_k + T\omega_k \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ v_k \\ \omega_k \\ \omega_k^b \\ 0 \\ \vdots \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ m_k^1 \\ \vdots \\ m_k^j \end{pmatrix} + \mathbf{w_k} \tag{7.11}$$

for readability we get

$$\begin{aligned} \mathbf{x}_k^{\mathrm{n}} &= \begin{pmatrix} x_k \\ y_k \\ \psi_k \end{pmatrix} \\ \mathbf{x}_k^{\mathrm{l}} &= \begin{pmatrix} v_k \\ \omega_k \\ \omega_k^b \end{pmatrix} \\ \mathbf{m}_k &= \mathbf{m}_k \end{aligned} . \tag{7.12}$$

To implement the particle filter, we restructure the combination of 7.11, 7.8 and 7.9 as according to [31, p.2 eq.8]

$$\begin{aligned} \mathbf{x}_{k+1}^{\mathrm{n}} &= \mathrm{f}_k^{\mathrm{n}}(\mathbf{x}_k^{\mathrm{n}}) + \mathrm{A}_k^{\mathrm{n}}(\mathbf{x}_k^{\mathrm{n}})\mathbf{x}_k^{\mathrm{l}} + \mathrm{B}_k^{\mathrm{n}}(\mathbf{x}_k^{\mathrm{n}})\mathbf{v}_k^{\mathrm{n}} \\ \mathbf{x}_{k+1}^{\mathrm{l}} &= \mathrm{f}_k^{\mathrm{l}}(\mathbf{x}_k^{\mathrm{n}}) + \mathrm{A}_k^{\mathrm{l}}(\mathbf{x}_k^{\mathrm{n}})\mathbf{x}_k^{\mathrm{l}} + \mathrm{B}_k^{\mathrm{l}}(\mathbf{x}_k^{\mathrm{n}})\mathbf{v}_k^{\mathrm{l}} \\ \mathbf{m}_{k+1} &= \mathbf{m}_k \\ \mathbf{y}_{1,\mathrm{k}} &= \mathrm{h}_{1,\mathrm{k}}(\mathbf{x}_k^{\mathrm{n}}) + \mathrm{C}_k(\mathbf{x}_k^{\mathrm{n}})\mathbf{x}_k^{\mathrm{l}} + \mathbf{e}_{1,\mathrm{k}} \\ \mathbf{y}_{2,\mathrm{k}}^j &= \mathrm{h}_{2,\mathrm{k}}(\mathbf{x}_k^{\mathrm{n}}) + \mathrm{C}_{\mathrm{j,k}}(\mathbf{x}_k^{\mathrm{n}})m_k^{\mathrm{j}} + \mathbf{e}_{2,\mathrm{k}}^{\mathrm{j}} \end{aligned} , \tag{7.13}$$

where $\mathrm{j} = 1, 2, \ldots, M_k$ are the known landmarks. To get the landmark measurements to

fit this model, the measurements can be inverted as [32, p.280, eq.30] and thus getting

$$\begin{aligned}
\mathbf{x}^{\mathrm{n}}_{k+1} &= \begin{pmatrix} x_k \\ y_k \\ \psi_k \end{pmatrix} + \begin{pmatrix} T\cos\psi_k & 0 & 0 \\ T\sin\psi_k & 0 & 0 \\ 0 & T & 0 \end{pmatrix} \begin{pmatrix} v_k \\ \omega_k \\ \omega^b_k \end{pmatrix} + \mathbf{w}^n_k \\
\mathbf{x}^{\mathrm{l}}_{k+1} &= \mathbf{0}_{3\mathrm{x}3} + \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_k \\ \omega_k \\ \omega^b_k \end{pmatrix} + \mathbf{w}^l_k \\
\mathbf{m}_{k+1} &= \mathbf{m}_k \\
\mathbf{y}_{1,\mathrm{k}} &= \begin{pmatrix} x_k \\ y_k \\ 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} v_k \\ \omega_k \\ \omega^b_k \end{pmatrix} + e_{1,k} \\
\mathbf{y}^j_{2,\mathrm{k}} &= -\mathrm{R}(\psi_k) \begin{pmatrix} x_k \\ y_k \end{pmatrix} + \mathrm{R}(\psi_k) m^j_k + e_{2,k}
\end{aligned} \quad . \tag{7.14}$$

To solve the MPF-SLAM problem, equation [31, algorithm 1 p.3] is used. As each particle is independent from other particles, parallel programming allows full use of the CPU. Most of the steps in marginalized particle filter are parallelised with help of OpenMP. The section of code that is to be run in parallel is marked with a preprocessor directive(pragma) that will cause new worker threads to spawn. This statement will also tell OpenMP how shared variables will be treated. For example the private directive tells OpenMP a given variable isn't shared between threads even if they share the same context(curly brackets). The `firstprivate` directive tells OpenMP to copy the given variable to the context opposed to `private` that only allocates the memory but leaves the variable uninitiated.

Example on a OpenMP loop parallelisation of a Kalman filter measurement update

```
void MarginalizedParticleFilter::kalmanFilterMeasurementUpdate(Eigen::
    MatrixXd y_ktmp)
{
  Eigen::MatrixXd x_tmp = Eigen::MatrixXd(nrStates,1);
  Eigen::MatrixXd htmp(nrMeasurements,1);
  Eigen::MatrixXd C_k_t = C_k.transpose();

  #pragma omp parallel for firstprivate(x_tmp, htmp, y_ktmp)
  for(int i = 0; i < N; i++)
  {
    // Get the states
    x_tmp = xi_k.col(i);

    // Calculate current function values for the measurement model and
       its jacobian
    measureModel.updateMeasureFunction(x_tmp, htmp);
    Eigen::MatrixXd htmp2 = htmp;
    RemoveRows(htmp2,inactiveMeasurements);

    // Calculate innovation
    Eigen::MatrixXd epsilon_k;
    epsilon_k = y_ktmp - htmp2;

    // Create a modified R_k matrix with removed columns and rows
    Eigen::MatrixXd R_prim = R_k;
    RemoveRowsAndColumns(R_prim, inactiveMeasurements);

    // Calculate innovation covariance
    Eigen::MatrixXd S_k;
    S_k = C_k*pi_k[i]*C_k_t + R_prim;

    // Calculate Kalman gain
    Eigen::MatrixXd K_k;
    K_k = pi_k[i]*C_k_t*S_k.inverse();

    // Calculate current state
    Eigen::MatrixXd stateFeedback = K_k*epsilon_k;
    xlfi_k1.col(i) = xlfi_k.col(i) + stateFeedback;

    // Calculate current covariance
    Eigen::MatrixXd covarianceFeedback = K_k*S_k*K_k.transpose();
    pi_k1[i] = pi_k[i] - covarianceFeedback;
  }
}
```

The pragma statement tells OpenMP to parallelise the loop and share the update steps between the threads, each thread is given it's own context(between the curly brackets) together with the variables explicitly defined in the private and firstprivate directive. All other variables is shared between the threads so extra caution have to be given so a variable isn't changed by multiple threads. The loop variable i is private for all threads as this variable is used to split the work between threads. So thread one might update all even indexes and thread two all odd. There are also extra directives such as scheduling,

but we expect each update takes roughly the same time thus the scheduling only introduce extra overhead. The number of threads spawned is automatically handled by OpenMP. Note that Eigen has OpenMP parallelisation built in, this has to be disabled when using OpenMP to parallelise Eigen calculations as we have done. OpenMP parallelisation is only relevant for larger matrices so there isn't any loss in our case.

To future optimise the calculations one possibility might be to use Intel's Math Kernel Library which will optimise the matrix calculations for every specific Intel processor. Eigen has the ability to link against this library thus there isn't any need to rewrite any code.

It should be noted we did not have time to implement positioning with regards to the laser scanner, but the skeleton from [31, p. 3, algorithm 1] is implemented as C++ code. Time didn't allow us to locate with regards to GPS-measurements. A code skeleton for this sensor is included. A distribution of this sensor is included from previous year but is not translated to our code standard.

The tuning parameters of this filter are the number of particles, the covariance matrix and the measurement noise. The first two are tuned in `MarginalizedParticleFilter.cpp` and the last is tuned in `MeasureModelNbaa.cpp`.

Issues and ways forward The filter is sensitive and will diverge if the matrices are not tuned carefully. Part of the problem is the numerical issues of the covariance matrix. If this gets too small, inverting results in nan (not a number). As for numerical stabilization, we recommend implementing the square root algorithm for the Kalman steps. This should improve the stability of the covariance matrix greatly taking both the eigenvalues and the symmetry into account. Another improvement could be scaling the states or using a solver instead of inverting a matrix.

We believe the filter will be more stable when more measurements are available. The filter is heavily affected by drift now as there is no measurement correcting displacement in position. This will handled mainly by the laser sensor, as detecting and associating objects will give a measurement of the position and also by the GPS-sensor.

7.3 Mapping

This part is only theoretical and is not implemented on the balrog.

The mapping subsystem is responsible to create a map of the surroundings of Balrog. It is also responsible to determine the probability that the whole area has been visited. Its task is to map the search area and to identify and classify objects from the point cloud.

The subsystem takes state estimates of the position from the positioning subsystem and information of detected obstacles from the obstacle detection subsystem as input. With this information the subsystem will create a probability map and also a map of the search area containing all obstacles.

7.3.1 Obstacle map

The obstacle map is an object containing information of were all detected obstacles are positioned. This map is dynamic and will come to change when new information is added

from the other subsystems.

The obstacles will be represented in a grid, with a resolution which is high enough. The grid will contain information whether a specific cell is unsearched, free or occupied with obstacle.

The grid will be updated when obstacles (lines) are discovered.

7.3.2 Probability map

The concept of the probability map is to give each square in the grid a probability that the square has been explored. The path finding subsystem needs this map to determine if a square needs to be revisited, that is whether the probability for the square having been explored is too low. The probability that a square (i, j) has been explored is updated accordingly to

$$\begin{array}{rcl} p_{1:k}(i,j) & = & 1 - (1 - p_k(i,j))(1 - p_{1:k-1}(i,j)) \\ p_k(i,j) & = & p(|x - r(i,j)| < L) \end{array} \tag{7.15}$$

where $p_{1:k}(i, j)$ is the probability that the square (i, j) has been explored up to time update k, $p_k(i, j)$ is the probability that Balrog is sufficiently close to the center of the square at the given time sample, $p_k(i, j)$ is the probability that the square was explored up to the previous time sample and $r(i, j)$ is the position of the center of square (i, j). The probability density function is computed as

$$f_{X,Y}(\mathbf{x}^{pos}) = \sum_i \pi^i \delta(\mathbf{x}^{pos} - x^i) \tag{7.16}$$

The probability is updated for the squares (i, j) which are in the perimeter of Balrog and is determined by the grid size. The probability is finally given by summing the weight of each particle within a circle, with radius L, around each square center, described as

$$\sum_k \Pi_1^k : \left|\mathbf{x}^k - r\left(i,j\right)\right| \leq L \tag{7.17}$$

This model assumes that the current position is independent of previous measurements which is a false assumption, but will work as an approximation.

8 Route Planning

The task of the route planning subsystem is to provide waypoints to the automatic control subsystem. The provided waypoints ensures that the whole searchable area is explored and mapped. The control subsystem marks waypoints as reached by signaling the route planning. The active waypoint can be modified by the route planing subsystem as the control subsystem will fetch the active waypoint each iteration in the control loop. All the calculations like `ObstacleInFront` and `FullyDiscovered` will happen in the route planing thread. The `NewCellsDiscovered` calculation will be done by checking if the discrete map data structure have a changed time stamp since last time there was a discovery.

8.1 Search algorithm

The search algorithm is the same as last years with the exceptions of a finer grid system as specified in [1] and a different obstacle following algorithm, see section 8.2.

The A*-algorithm is used to find the optimal path and it consists mainly of two parts. The goal function calculate the cost from balrog current position to the goal and the value is the euclidian distance between the two points. The path-cost function is determined by several things. The cost depends on the previously path-cost, if Balrog has to make turns, if cells are adjacent to obstacles and if a point is on a straight line between the start and goal position. The optimal path is the one with lowest total cost.

When the robot is initiated there will not be any initial waypoints until the operator inputs a search area. When this is done, a list of waypoints are generated so the whole area can be explored. This will be accomplished by placing the waypoints in a zig-zag pattern. Since the squares in the grid are smaller than Balrog, waypoints will be placed with sufficiently intermediate spacing. See Figure 8.1. The flow is described below and in Figure 8.2.

1. Obstacle in front? If yes go to *Obstacle following mode* (see Section 8.2), if no go to 2.

2. Reached final way point? If yes, Balrog will revisit squares that are unlikely to have been searched according to the probability map, see section 7.3.2, using the A* algorithm and then stop at the final position. This will also set the route planning in a mode waiting for new user interaction. If we have not reached the final waypoint go to 1.

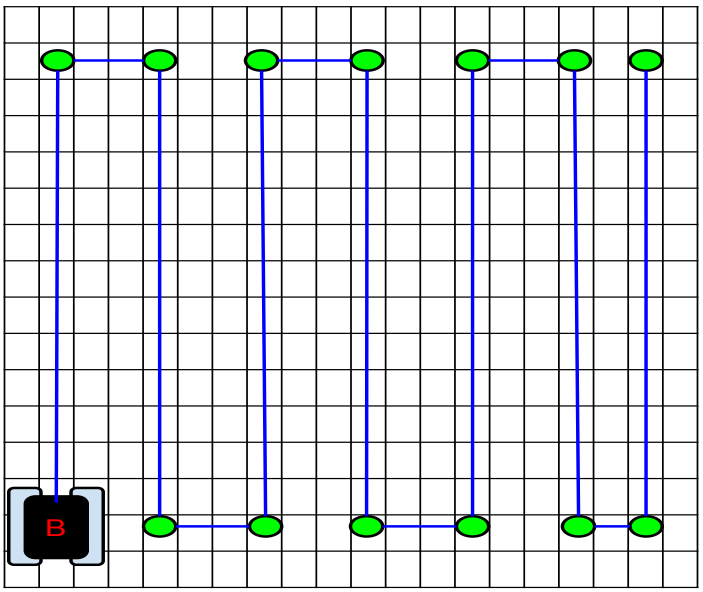

Figure 8.1: Zig-zag search pattern. Green dots are waypoints, blue lines are reference route for Balrog to follow

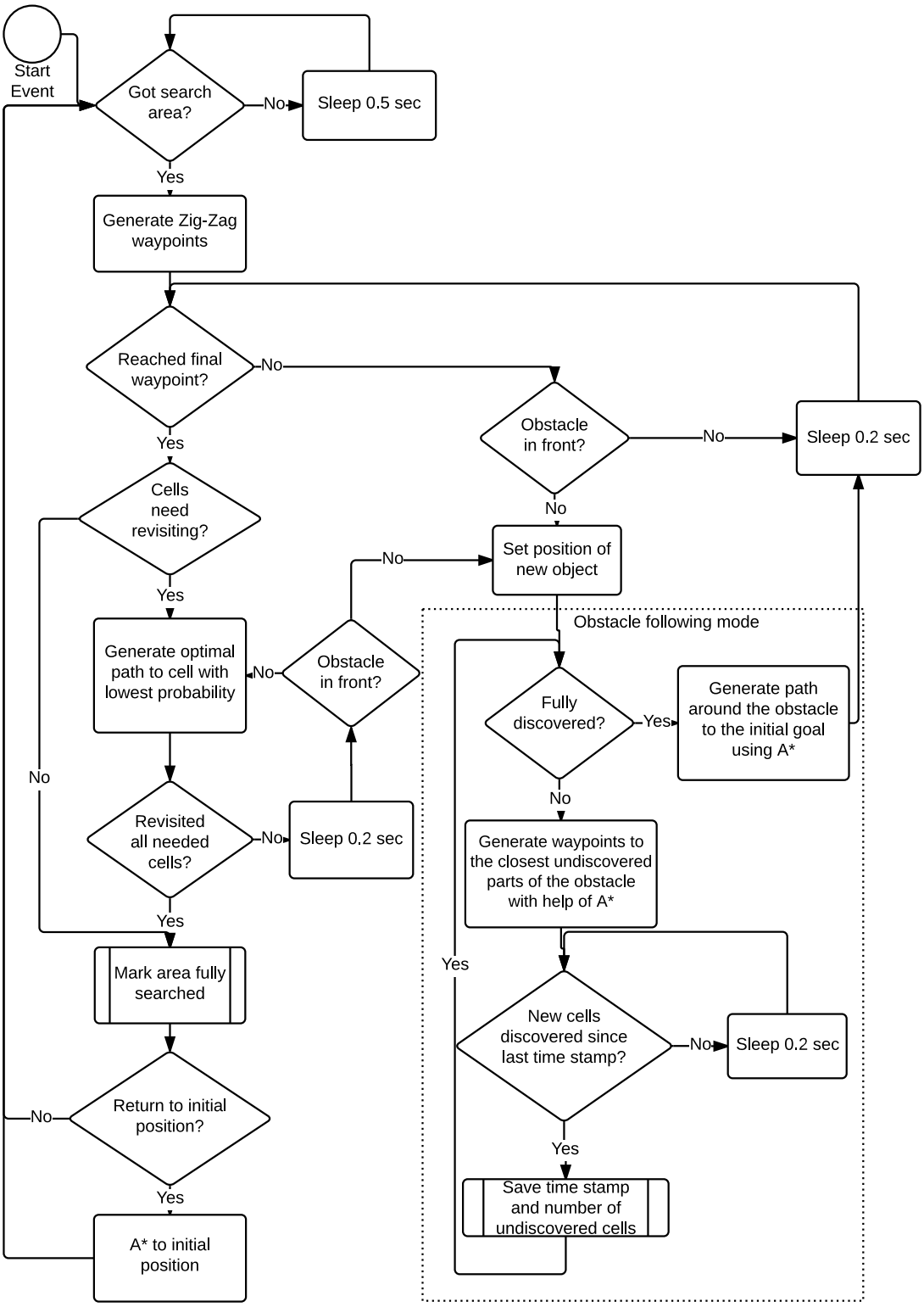

Figure 8.2: Program flow of the route planning sub system

8.2 Obstacles

To navigate around the obstacles, the route planning system will create waypoints for the automatic control system to follow. Waypoints are updated when the route planning notes that there is a obstacle in front of the robot if the robot is in waypoint mode and when new cell (parts of obstacles in their discrete form) is updated if we are in the obstacle following mode.

In waypoint mode the route planing subsystem is sleeping waiting for obstacles to be in front of the robot, then checks the gridded obstacle map if it has been updated. If so, the route planing subsystem changes to *Obstacle following mode*. The system flow of the *Obstacle following mode* is described below.

1. Generate waypoints to the closest undiscovered part of the obstacle by using the A* algorithm. If multiple undiscovered points with the same straight line cost is found, the A* algorithm is run multiple times. If there are no undiscovered cells around the obstacle, generate a new path using A* algorithm to the original (the initial goal before there was a obstacle in front of the robot) waypoint

2. Wait for new cells to be discovered. When new cells have been discovered go to 1

The grid have a greater resolution than previous year to make it possible to navigate around the irregular obstacles smoothly. The A*-algorithm will take this into account when calculating the path around an obstacle. If the a grid is smaller than 50cm, then a penalty is added to grids adjacent to obstacles. Hence, the A*-algorithm will not choose these grids in the path and the balrog will not collide with obstacles.

Note: If the grid size is smaller than 20 cm this will have to be taken into account and the algorithm has to be modified. Adding penalty cost to more grids than just the ones adjacent to obstacles.

The mapping subsystem in SLAM contains and generates the gridded map including margins, see Section 7.3.1.

9 Mine detection

The mine detection subsystem is responsible for detecting the mines surrounding Balrog. The mines are represented as magnets and are detected using a magnetometer, which is inside the IMU. The algorithms and implementation of the mine detection will not be changed from last year's project and will run in a separate thread. This means that a mine is detected if the magnetometer is saturated, which occurs at 1.2 Gauss. This value should be set in comparison to Earth's magnetic field strength at 0.5 Gauss.

The mine is considered new if the distances between Balrog and already explored mines are sufficiently large. If a new mine is detected, the position of the mine is set to the estimated position of Balrog. Balrog is able to detect mines in a radius of approximately 0.5 meters and has no information regarding what direction from Balrogs view point the mine is placed. Either we need to implement a search algorithm to estimate the position better or use multiple magnetometers to triangulate the position. But as this is outside the scope for this project, we have decided to leave the implementation used last year. The mine detection is implemented, however it is not tested if it operates as desired.

10 Automatic control

The controller part will be mostly unmodified from last year's project [20]. The controller will aim to go straight to the next waypoint, with help of the waypoint list created by the route planner and the position and bearing estimate provided by the SLAM algorithm. It will do so by trying to minimize the difference between the actual bearing ψ_k and the bearing ψ_k^* required to reach the next waypoint. A flow chart of the control algorithm when close to a waypoint is shown in Figure 10.1 and described further in Section 10.1. The variables used is shown in Table 10.2.

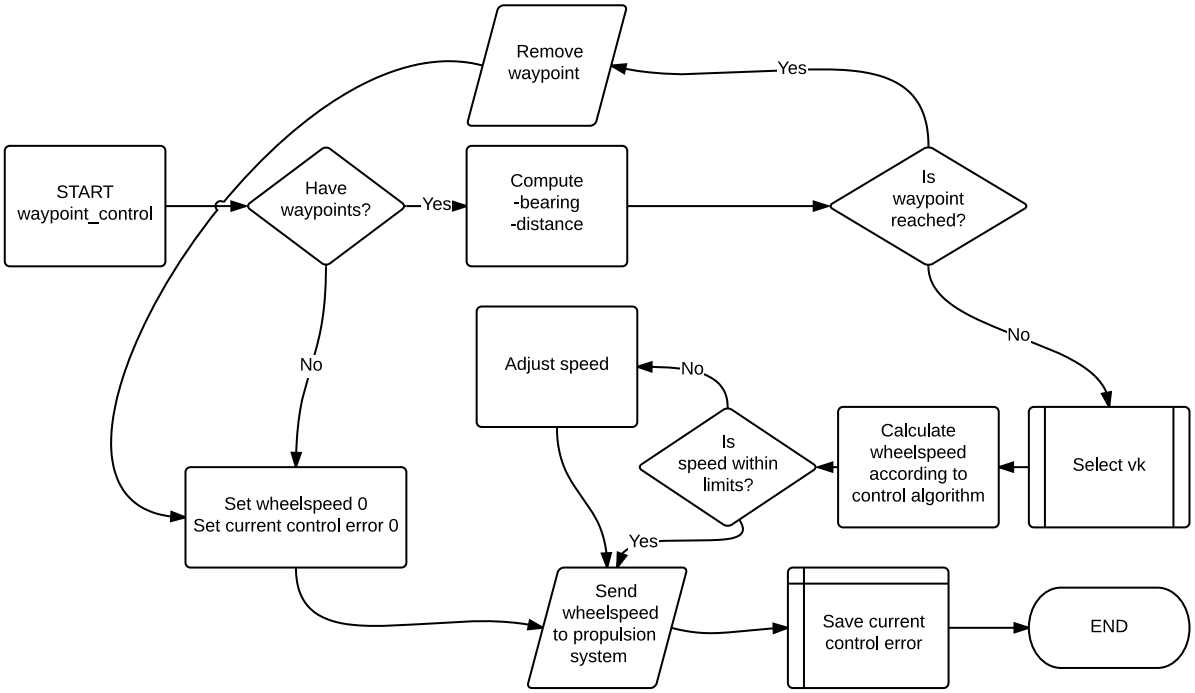

Figure 10.1: Flow of control algorithm.

10.1 Control equations

The controller consists of two parts, v_k and v_{corr}. Here, v_k is the desired speed and v_{corr} is the PD controller according to

$$v_{\text{corr}} = K\varepsilon_k + \frac{T_d}{T_s}(\varepsilon_k - \varepsilon_{k-1}). \tag{10.1}$$

Table 10.2: Description of control variables.

Variable	Description
$x_{k,w}$	x coordinate of next waypoint at time instant k (comes from route planner)
$y_{k,w}$	y coordinate of next waypoint at time instant k (comes from route planner)
x_k	x coordinate of estimated position of Balrog at time instant k (comes from SLAM)
y_k	y coordinate of estimated position of Balrog at time instant k (comes from SLAM)
ψ_k	Estimated bearing at time instant k (comes from SLAM)
ψ_k^*	Bearing required to reach the waypoint from current position at time instant k
ε_k	Bearing error at time instant k
d	Distance between Balrog and the next waypoint at time instant k
V_{max}	Maximum allowed speed
V_{min}	Minimal allowed speed
d_{lim}	The radius around a waypoint where Balrog moves slowly and more accurate
$d_{reached}$	How close Balrog must be to a waypoint for it to be considered reached
$\varepsilon_{k,max}$	Maximal allowed bearing error
K	Proportional gain constant
T_d	Derivative time
K_d	Derivative gain constant
α	Scaling factor
v_{corr}	PD controller
v_k	Desired speed
$v_{l,k}$	Speed of left tack
$v_{r,k}$	Speed of right track

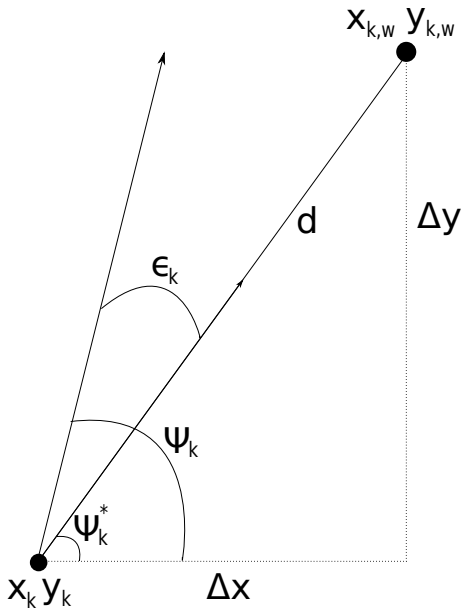

Figure 10.2: Angle error

The bearing error ε_k can be calculated as

$$\begin{aligned} \Delta x_k &= x_{k,w} - x_k, \\ \Delta y_k &= y_{k,w} - y_k, \\ d &= \sqrt{\Delta x_k^2 + \Delta y_k^2}, \\ \psi_k^* &= \text{atan2}(\Delta x_k, \Delta y_k), \\ \varepsilon_k &= \psi_k^* - \psi_k, \end{aligned} \tag{10.2}$$

and the connection between the variables can be seen in Figure 10.2. v_k is given by

$$v_k = \begin{cases} 0 & \text{if } \varepsilon_k \geq \varepsilon_{\text{k,max}}, \\ \min(\frac{V_{\max}}{1.2}, K_d d + V_{\min}) & \text{if } d \leq d_{\text{lim}} \text{ and } \varepsilon_k < \varepsilon_{k,\max} \text{ or} \\ \alpha(V_{\max} - |V_{\text{corr}}|) & \text{if } d > d_{\text{lim}} \text{ and } \varepsilon_k < \varepsilon_{k,\max}, \end{cases} \tag{10.3}$$

and as shown, calculated differently depending on how close to the waypoint the Balrog is and how big the angle error is. α is a scaling factor that can be used to reduce the speed.

In Figure 10.3 a flow chart for choosing v_k is shown.

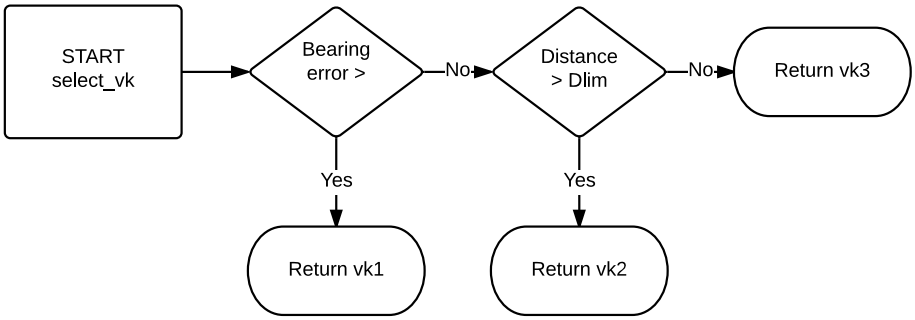

Figure 10.3: How to choose correct v_k.

The speed for the left track, $v_{l,k}$ and right track $v_{r,k}$ is computed by subtracting v_{corr} from the left track and adding it to the right, according to

$$\begin{aligned} v_{l,k} &= v_k - v_{\text{corr}} \\ v_{r,k} &= v_k + v_{\text{corr}} \end{aligned} \tag{10.4}$$

with a rotation as a result. Notice that $|v_{l,k}|$ and $|v_{r,k}|$ must be in the range $[V_{\min}, V_{\max}]$ for Balrog to move at all.

A waypoint is considered reached if $d < d_{\text{reached}}$. In that case the waypoint is removed and the next waypoint is used instead.

10.2 Proposed values of constants

Table 10.3 lists last project's references of constants and limits [20]. They have not been validated but should be a good starting point.

Table 10.3: Table of constants and limits.

Constant	Proposed value
V_{max}	760 mm/s
V_{min}	100 mm/s
d_{lim}	1 m
$d_{reached}$	0.3 m
$\varepsilon_{k,max}$	20°
α	1 (new this year)

11 Further Development

Since Invenire Periculosa did not quite reach the goal of having a fully operational system, this section describes what is left for completing the implementation, and our idea of how the complete product should be assembled. The following subsections describe a problem, what the current status is and (if we have an idea of implementation) what measures that should be taken for fixing the problems. There are also some minor todo:s left in the doxygen comments [16] that should be taken care of by the next group that develop the Balrog further.

11.1 Interaction between route planner and waypoints

At the moment both the controller and route planner has access to the shared data structure called `waypoints`. Waypoints holds waypoints that describes the route Balrog is supposed to take. When the controller reaches a waypoint it removes it from the shared waypoints structure.

When the route planner runs it sets a variable called `waitForWaypoints` in `waypoints`. The route planner then rearranges the waypoints and assumes that nobody else reads or writes to `waypoints` during this. In other words `waitForWaypoints` has the semantic meaning of a mutex, but since it s a boolean this is not enforced. If the current structure where the routeplanner "locks" the `waypoints` structure it should be implemented using a real mutex rather than a boolean flag.

Another way to see it is that `waypoints` always contains a list of active updated waypoints, when a new obstacle is detected, the routeplanner can remove all waypoints from `waypoints` (since they are no longer valid). It then calculates new waypoints and add them back to `waypoints`. This way there is no risk that other subsystems access `waypoints` and retrieves invalid waypoints (since they will see an empty `waypoints` structure).

The route planner also adds waypoints from the back. The reason for this is that the A*-algorithm needs to search down a node-tree to find the best path. Hence, it cannot start placing waypoints before it has found the last node in the tree. It is also more convenient to add items to the back of a vector instead of in the front.

11.2 Interactions between threads

All sharing of data between threads (subsystems) are currently implemented using shared data structures and mutually exclusive locks. Some communication might benefit from other constructs such as conditional variables or some message passing structure between threads.

Currently there is no way for a thread to be notified if a change in a shared data structure occurs, a message passing setup would eliminate the overhead of threads periodically checking if the shared data structure has changed.

11.3 Integration of basestation with new Balrog

Currently a very limited set of commands in the basestation are actually fully implemented. See section 4 for a list of what is implemented. There is also a need for integrating the new implementation to the GUI. At the moment the basics are roughly implemented in the GUI. We recommend that next year's project should focus on "Extreme GUI makeover"^{TM} to create a simpler source code for the GUI so future implementations can be more easily integrated.

11.4 Route planner

Currently the route planning subsystem and related thread is not executed in the main file on the Balrog. The reason for this is that the Large loop is not fully implemented and tested. In the file `RoutePlanner.cpp` "*@todos*" are found which show what is needed to implement to make the subsystem work. Since not everything is finished, an unit test have to be made to check if everything work as intended. To fully understand how the route planning works, check the flow chart in Figure 8.2. The route planning subsystem operates mainly on the obstacle and probability map and is highly dependent on that these maps are fairly accurate.

11.5 Mine detection

The mine detection is implemented and works theoretically. Due to lack of time, this functionality has not been tested in practice with real mines (magnets). This should be tested and evaluated to ensure good performance.

11.6 SLAM

Localization To locate we use three measurements: angular velocity from the IMU and speed and turning rate from the odometers (corresponds to the last three equations in 7.8. A GPS measurement is ready to be used and is a mere calculation of the weight to be implemented. The GPS isn't accurate, so implementing this sensor wasn't of high priority when time was running out. It should be accurate enough to compensate long time drift.

When the landmarks are associated the measurements from the laser sensor should help cancelling the drift substantially. Computing well thought landmarks and extracting them with a good algorithm takes time so a big focus should be put here (see 7.1 for ways to do this).

Filter A marginalized particle filter has been implemented (see 7.2.4) to fuse the sensor measurements. The part that is left is the landmark measurement update and estimation [31, step 5 algorithm 1 p. 3]. A code snippet for [31, eq. 22-23 p.4] is written but untested and unvalidated. Consideration has to be taken on how to model the landmarks. There are also some numerical issues which should be taken care of. Again, see 7.2.4 for suggestions.

Obstacle detection The algorithm implemented in 7.1.1 discovers edge points between objects, but not from angles within the same region. To detect angles, a similar analysis to the algorithm used in 7.1.1 can be used, but instead of checking to differences in distances from the laser sensor, checking the derivate in the differences in the distances from the laser sensor.

Landmark association Associating landmarks isn't implemented and has to be done from scratch. Ideas and algorithms on how this could be done can be found here.

The lines could be matched to each other using an ICP-algorithm. The ICP (Iterative Closest Point)-algorithm compares a given point cloud (source) to a reference point cloud. It finds the closest corresponding point between the two clouds, then it estimates a transformation consisting of rotation and translation. This transformation is the one that optimally aligns the points according to their mean square error. The source is then transformed, and the procedure is iterated. The output is the transformation matrix. It finds the closest points matching between model and data. Singular Value Decomposition (SVD) is used to determine the optimal transformation.

There are two possible techniques, point-to-line and point-to-point matching.

To perform the point-to-point matching [33, Algorithm 1] could be implemented.

To perform point-to-line matching the algorithm needs two different laser scan rotations at nearby times (model and data), together with an initial guess of the transformation matrix describing the distortion. Lines are extracted by using RANSAC or split-and-merge (see 7.1.1). A skeleton in pseudo code for this algorithm already exists in the obstacle detection class. A version of the algorithm is described in [34, fig 5]. An example of an implementation can also be found in [35].

The unmatched points are then used to create lines using RANSAC or split-and-merge (see 7.1.1), RANSAC being already implented in our code.

Mapping The mapping part of SLAM is not implemented, section 7.3 describes how this should be done.

11.7 Hand controller

The hand controller's main purpose is to enable manual control of Balrog. However there are several additional functions the hand controller could be equipped with to increase the usability. Function that could be implemented in the hand controller are for instance:

- Set Balrog to automatic mode
- Set Balrog to manual mode
- Start search
- Stop Balrog (Emergency break)
- Reset search
- Set the origin to Balrog's current GPS coordinates

In addition to this, it could be interesting to make the hand controller vibrate when Balrog locates a mine.

References

[1] Marcus Bäck. *Requirement specification*, September 2014. http://www.isy.liu.se/edu/projekt/tsrt10/2014/bandvagn/Documents/Requirement_specification_v1.0.pdf.

[2] Free Software Foundation. *GCC 4.6.3 manuals*. https://gcc.gnu.org/onlinedocs/4.6.3/.

[3] Ubuntu. *Ubuntu 14.04.1 LTS (Trusty Tahr)*. http://releases.ubuntu.com/14.04/.

[4] Digia Plc. *Qmake manual*. http://qt-project.org/doc/qt-4.8/qmake-manual.html.

[5] Cmake. *Cmake)*. http://www.cmake.org/.

[6] Autoconf. *Autoconf)*. https://www.gnu.org/software/autoconf/.

[7] Digia Plc. *Download Qt*. http://qt-project.org/downloads.

[8] *Getting Started with GIT*. http://git-scm.com/book/en/v1/Getting-Started.

[9] Google. *Google-glog: Logging library for c++*. https://code.google.com/p/google-glog/.

[10] Google. *googletest: Testing library for c++*. https://code.google.com/p/googletest/.

[11] Eigen webpage. *Eigen*. http://eigen.tuxfamily.org/index.php?title=Main_Page.

[12] Robo Peak Team. *RPLIDAR A1M1-R1 development kit*. http://rplidar.robopeak.com/download.html.

[13] Ubuntu. *Paket: build-essential (11.6ubuntu6)*. http://packages.ubuntu.com/trusty/build-essential.

[14] Marcus Bäck. *User manual*, December 2014. http://www.isy.liu.se/edu/projekt/tsrt10/2014/bandvagn/Documents/user_manual_v1.1.pdf.

[15] *Travis CI for private repositories*. https://travis-ci.com/.

[16] Invenire Periculosa. *Doxygen automatic generated comments for Balrog 2014*, December 2014. http://www.isy.liu.se/edu/projekt/tsrt10/2014/bandvagn/doxygen/index.html.

[17] Hartmut Schimmel. *Connecting AT91xx based GPS-Loggers to USB on Linux*. http://www.schimmelnetz.de/projekte/iTU4l/usb.html.

[18] *Persistent names for usb-serial devices*. http://hintshop.ludvig.co.nz/show/persistent-names-usb-serial-devices/.

[19] *Best way to set serial port speeds on boot?* http://unix.stackexchange.com/questions/91458/best-way-to-set-serial-port-speeds-on-boot.

[20] Emmeline Kemperyd. *Teknisk Dokumentation Minröjningsbandvagn*, December 2013. http://www.isy.liu.se/edu/projekt/tsrt10/2013/bandvagn/teknisk_dokumentation.pdf.

[21] i2cchip.com. *BL233B, RS232 I2C/SPI/1WIRE ADAPTOR & CONTROLLER*, October 2014. http://www.i2cchip.com/pdfs/bl233_b.pdf.

[22] u-blox AG. *u-blox 5 Receiver Description*, November 2009. http://www.u-blox.com/images/downloads/Product_Docs/u-blox5_Protocol_Specifications(GPS.G5-X-07036).pdf.

[23] Robo Peak Team. *RPLIDAR Introduction and data sheet*, May 2014. http://rplidar.robopeak.com/download.html.

[24] Silicon Labs. *Serial Communications Guide for the CP210x*. http://www.silabs.com/Support%20Documents/TechnicalDocs/an197.pdf.

[25] Robot Peak Team. *RPLIDAR Interface Protocol*, Mars 2014. http://rplidar.robopeak.com/download.html.

[26] Micro Strain. *3DM-GX1 Gyro Enhanced Orientation Sensor Technical Overview*, 3.1.02 edition, September 2014. http://files.microstrain.com/3DM-GX1%20Datasheet%20Rev%201.pdf.

[27] Micro Strain. *Detailed Specifications For 3DM-GX1*, September 2014. http://files.microstrain.com/3DM-GX1%20Detailed%20Specs%20-%20Rev%201%20-%20070723.pdf.

[28] Micro Strain. *3DM-GX1 Data Communications Protocol*, 3.1.02 edition, September 2014. http://files.microstrain.com/manuals/3DM-GX1%20Data%20Communication%20Protocol%203102.pdf.

[29] Agostino Martinelli Nicola Tomatis Roland Siegwart Viet Nguyen, Stefan Gächter. *A comparison of line extraction algorithms using 2D range data for indoor mobile robotics*. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1545234.

[30] Konstantinos G. Derpanis. *Overview of the RANSAC Algorithm*, 2010. http://www.cse.yorku.ca/~kosta/CompVis_Notes/ransac.pdf.

[31] David Törnqvist Fredrik Gustafsson Thomas B Schön, Rickard Karlsson. *A Marginalized Particle Filtering Framework for Simultaneous Localization and Mapping*, 2007. http://users.isy.liu.se/en/rt/fredrik/reports/07fusionfastMPFslam.pdf.

[32] Fredrik Gustafsson. *Statistical Sensor Fusion*, 2012.

[33] Seungpyo Hong. *Improved Motion Tracking with Velocity Update and Distortion Correction from Planar Laser Scan Data*, 2008. http://www.imrc.kist.re.kr/MediaWiki/images/3/35/laser_tracking.pdf.

[34] Andrea Censi. *An ICP variant using a point-to-line metric*, May 2008. `http://censi.mit.edu/pub/research/2008-icra-plicp.pdf`.

[35] Andrea Censi. *Canonical Scan Matcher*. `http://censi.mit.edu/software/csm/`.

Appendices

A Coding standard

The goal of coding standards is to increase the business value of the code. The most obvious (and indeed most important) way to do this is to make the code robust and correct. Equally important, but more subtle goals include reducing coder friction and maintainability.

The aim of this coding standard document is to keep it short and simple, therefore it might not cover all possible situations. If an undocumented situation appears, bring it up for discussion and add it to the standards!

Good code is

- Straightforward, not clever
- Understandable to somebody who has not seen the code before
- Consistent
- Well documented

General rules

- **Eliminate duplicate code** - Any code that is used twice must be refactored out and put in a common class
- **Choose great names** for your classes, methods and variables - they should be self descriptive rather than short
- **Comment the what and why** rather than the how, the code in itself should be self explanatory, comments should not duplicate what the code says
- **Declare variables** in the most limited scope possible
- **Prefer small and focused functions**
- **Prefer function overloading rather than default arguments**
- **Always include curly braces on conditional and loop statements**
- **Use explicit constructor**
- **No compiler errors or warnings** from the final code

Namespace

- Wrap all classes and objects in the namespace `Balrog`
- Do not use *using namespace* anywhere, it pollutes the global space.

Header files

Header files ends in *.h* and is named after the class or data structure inside it, for example *ClassName.h* and are placed in an *include* directory.

All header files must avoid header collision by wrapping the code in the header file

```
#ifndef BALROG_CLASS_NAME_H
#define BALROG_CLASS_NAME_H

...

#endif  // BALROG_CLASS_NAME_H
```

Declaration order within a class

- Public before private,
- Methods before data members (variables), etc.

Class definitions must start with its public section, followed by its protected section and then its private section. If any of these sections are empty, omit them.

Within each section, the declarations must be in the following order:

- Typedefs and Enums
- Constants (static const data members)
- Constructors
- Destructor
- Methods, including static methods
- Data members (except static const data members)

Source files

All source files ends with *.cpp* and are placed in a *src* directory.

Naming formats

- **Classes:** Camelcase, i.e. `ClassName`
- **Abstract classes:** Prefix classname with A, i.e. `AClassName`
- **Interfaces:** Prefix classname with I, i.e. `IClassName`
- **Variables:** Lower camel case, i.e. `myVariable`

- **Member functions:** Lower camel case, i.e. `myFunctionName()`
- **Constants:** Upper case and underscores, i.e. `I_AM_GLOBAL`
- **Getters and setters**: Lower camel case with get/set prefix, i.e. `getMyVariable`
- **Private variables**: Postfix underscore, i.e. `myPrivateMember_`

Initialize variables upon creation

```
int i=10;  // This is correct

int i;
i=10;   // This is wrong
```

Use of const

- All member functions not altering the state of the object must be marked const
- All member variables that do not change during the lifetime of the object must be marked const
- All method parameters not modified in the method must be marked const

Memory semantics

The use of pointers and references is sometimes error prone. Who is responsible of cleaning up and freeing memory?

- Passing a pointer to an object or method means that this object is responsible for cleaning it up
- Passing a reference means that the caller retains ownership

Inheritance

Classes should extend from at most one class that is not abstract or an interface. There might be exceptions to this rule, but think carefully. Twice.

Globals

Never use globals. There might be exceptions to this rule, but think carefully. Twice.

Comments

Comments are written in Doxygen javadoc style. All classes and methods **MUST** be documented. All high level comments are kept in the header files. Only short implementation specific comments are in the cpp files.

Classes

Include at least a general description of the class and its purpose.

Methods

At least the purpose of the method along with input and output must be documented, also include information about exception and error handling.

Automatically generated constructors and operators

Remove copy constructors and copy operators if they are automatically generated but render unexpected behaviour. Always implement them if it makes sense.

Source format

- Use tabs, 2 spaces wide
- Namespaces do not add an extra line of indentation
- Line width: 80

Conditionals

```
if(condition)
{  // spaces inside parentheses - rare
  ...
}
else
{
  ...
}
```

Loops

```
for(int i = 0; i < kSomeNumber; ++i)
{
  printf("i");
}
```