Design Specification Minesweeper

Version 1.0

Editor: Elin Näsholm
Date: October 31, 2017

Status

Reviewed	Elin Näsholm	11/10-17
Approved	Martin Lindfors	12/10-17

Course name:	Automatic Control - Project Course	Course code:	TSRT10
Project group:	Smaugs	E-mail:	tsrt10-minrojning@googlegroup.com
Project:	Minesweeper	Document name:	Design Specification

Project Identity

Linköping University, Department of Electrical Engineering (ISY)
Autumn 2017

Client:	Martin Lindfors, Linköping University
	Phone: +46 13 28 13 65, **E-mail:** martin.lindfors@liu.se
Customer:	Torbjörn Crona, Saab Dynamics
	E-mail: torbjorn.crona@saabgroup.com
Course Examiner:	Daniel Axehill, Linköping University
	Phone: +46 13 28 40 42, **E-mail:** daniel.axehill@liu.se
Project Manager:	Hampus Andersson
Advisors:	Per Boström-Rost, Linköping University
	E-mail: per.bostrom-rost@liu.se
	Erik Ekelund, Saab Dynamics
	E-mail: erik.ekelund@saabgroup.com
	Axel Reizenstein, Saab Dynamics
	E-mail: axel.reizenstein@saabgroup.com

Group members

Name	LiU-id	Phone number	Responsibility
Andreas Hägglund	andha796	072-713 38 70	Head of Hardware
Andreas Lundgren	andlu901	070-022 56 44	Head of Design
Elin Näsholm	elina044	073-094 94 03	Head of Documentation
Fredrik Gustafsson	fregu856	070 578 63 48	Head of Slam Implementation
Fredrik Tormod	freto995	073-775 05 36	Head of Control and Route Planning
Hampus Andersson	haman657	073-675 93 56	Project Manager
Jonathan Jerner	jonje173	070 296 61 50	Head of Software
Mattias Andreasson	matan461	070-822 53 67	Head of Testing

Group E-mail: tsrt10-minrojning@googlegroup.com
Homepage: TBA

Mail to each individual can be sent to "LiU-id"@student.liu.se.

Contents

Document History

Version	Date	Changes made	Sign	Reviewer
0.1	18/9-17	First draft.	-	EN
0.2	25/9-17	Second draft	-	EN
0.3	27/9-17	First revision	-	EN
0.4	1/10-17	Second revision	-	EN
0.5	3/10-17	Third revision	-	EN
0.6	9/10-17	Fourth revision	-	EN
0.7	11/10-17	Fifth revision	-	EN

1 Introduction

This document is a design specification for the Minesweeper project in the course *TSRT10: Automatic Control - Project Course* at Linköping University. The purpose of the document is to provide an overview of the system and to describe its subsystems and functionality in detail.

The purpose of the project is to further develop an autonomous minesweeping system. The system shall be improved by implementing a SLAM algorithm utilizing a new scanning LIDAR sensor mounted on the ground vehicle (Balrog), for more accurate localization and mapping. The route planning and navigation of Balrog shall also be improved for more efficient exploration of unknown areas. A UAV (Sauron) shall be integrated in the system to assist Balrog with its minesweeping tasks. The long-term project goal is to develop a fully autonomous minesweeping system that is able to detect landmines safely, efficiently and accurately.

1.1 Definitions

AprilTags	A matrix of bar codes used to identify predefined objects.
Balrog	The ground vehicle aimed for minesweeping used in the project.
EKF	Extended Kalman Filter.
GUI	Graphical user interface.
HAL	Hardware abstraction layer.
MAV-link	Micro Air Vehicle Link.
Mine	An object or image on the ground acting as a mine.
Object	Any physical item in the testing area such as a mine, an obstacle or a wall.
Obstacle	An object in the test area that acts as an obstacle for the Balrog.
ROS nodes	Units of code that can be run independently of each other and communicate via ROS topics.
ROS topics	Named data channels that ROS nodes can send messages to by publishing on the topic, and read messages from by subscribing on the topic.
ROS	Robot Operating System. A set of open source software libraries and tools enabling communication between different modules in robotic systems.
Route	A set of way-points created by the user or the platform itself.
rqt	Qt-based framework for GUI development in ROS.
rviz	Visualization tool for ROS.
Sauron	The drone used in the project for collaborating with the ground vehicle.
Search area	The area that has not yet been searched for mines by Balrog.
Searched area	The area that has been visited by Balrog.
SLAM	Simultaneous localization and mapping.
TCP	Transmission Control Protocol.
Test area	An indoor predefined area where the platform will be tested.
The project group	The group of students involved in the project.
UAV	Unmanned Aerial Vehicle.
UDP	User Datagram Protocol.
User	The human interacting with the system in any way.

2 System Overview

The minesweeping system consists of three main subsystems: Balrog, Sauron and BaseStation. Balrog is a tracked robot equipped with various sensors and computing hardware, designed to autonomously explore and map an assigned area on the ground. Sauron is a UAV equipped with additional onboard computing hardware and a camera, designed to provide an aerial view of the assigned area and assist Balrog in performing its task. Both platforms can be manually controlled by a human user via RC hand controllers and they are both connected to BaseStaion over WiFi. BaseStation is a computer running ROS that communicates with the platforms and displays relevant sensor data in separate GUIs. An overview of the system is seen in Figure 1.

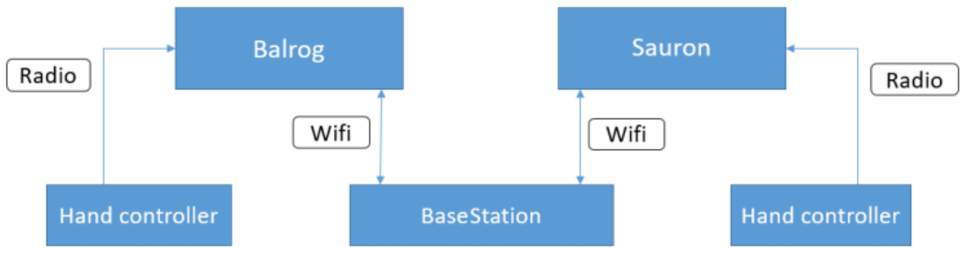


Figure 1: Schematic overview of the minesweeping system.

Each subsystem does in turn consist of a number of software modules implementing different key functionality.

Balrog consists of the following software modules:

SLAM	Estimation of the current Balrog position (localization) and map creation of the surrounding area including obstacles and visited areas (mapping).
Navigation	Decision making and computation of routes to explore the assigned area in an efficient way.
Control	Route following.
Communication	Communication between software modules.

Sauron consists of the following software modules:

Detection	Detection of Balrog in video frames from the onboard camera.
Tracking & Control	Controlling Sauron to physically track Balrog based on the detection.

BaseStation consists of the following software modules:

Balrog GUI	Visualization of the map and various sensor data, including a live stream from the camera onboard Balrog. Reading of user input.
Sauron GUI	Visualization of various sensor data, including a live stream from the camera onboard Sauron. Reading of user input.

The Balrog and Sauron subsystems are designed to be deployed together in order to explore an assigned area in the most efficient way possible. To simplify the development process and to ensure modularity, the subsystems are however designed to also function independently. In this mode, two separate base station computers can be utilized to run the corresponding GUI software.

3 Balrog

In this section the Balrog subsystem is described in detail. A high-level hardware schematic is presented, its main hardware components are listed and implementation details are provided for its different software modules.

3.1 Hardware

The main hardware component of Balrog is its tracked robot platform with motors and speed controllers, onto which all other components are mounted. The robot platform has been developed in previous years' projects and is considered as one integrated component. No further description of the tracked platform will be provided in this section, the focus will instead be on Balrog's sensor setup and computing hardware. An overview of the Balrog hardware is seen in Figure 2.

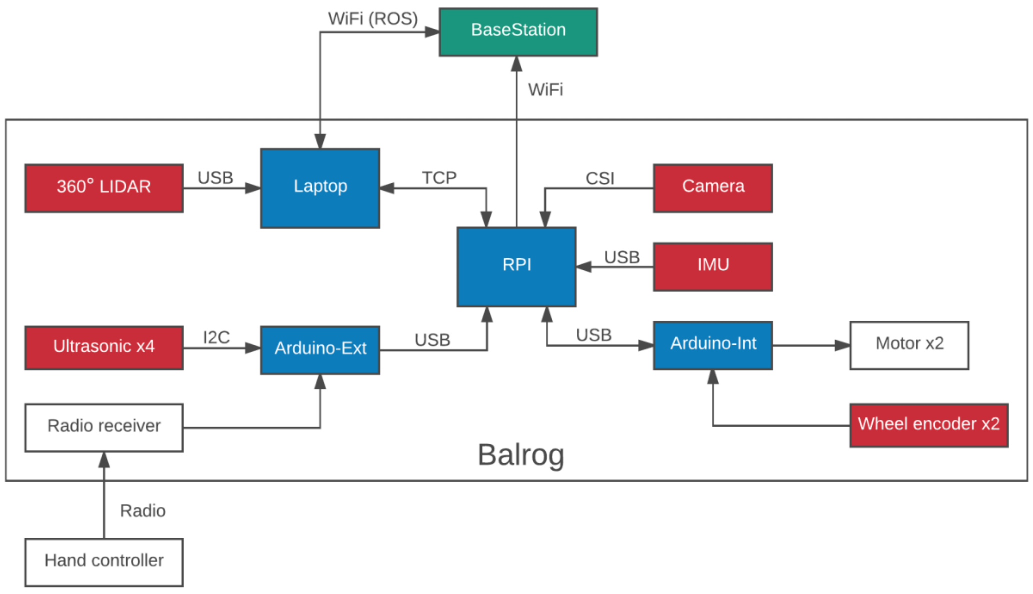


Figure 2: Schematic overview of the Balrog subsystem hardware.

3.1.1 Computing Hardware

Of the hardware components seen in Figure 2 above, the following constitutes Balrog's onboard computing capabilities:

Laptop	A laptop running ROS placed on the platform to run the computationally demanding SLAM module, to communicate with *RPI* (see below) and to communicate with BaseStation over WiFi.
RPI	A Raspberry Pi 3 model B+ that receives maps and an estimated robot pose from the SLAM module on *Laptop*, it runs the modules for navigation and control and communicates with the Arduinos (see below) to operate the tracked platform.

Arduino-Int	An Arduino that reads control signals from *RPI* and transforms these into motor commands which it sends to the platform motors. It also reads data from the wheel encoder sensors and forwards this to *RPI*.
Arduino-Ext	An Arduino used mainly for manual control of Balrog. It reads data from the RC hand controller and forwards this to *RPI*. It can also read data from the ultrasonic rangefinders via I^2C and forward this to *RPI*.

3.1.2 Sensors

Of the hardware components seen in Figure 2, the following constitutes Balrog's sensor setup. Data for some sensors are taken from the Technical Documentation of the 2016 Minesweeper project [1].

360 LIDAR	A Scanse Sweep v1 360° scanning LIDAR mounted on top of the robot platform. A range sensor that functions by transmitting a laser beam towards a target and detecting the reflected light. The sensor has a maximum range of 40 m, a rotation frequency of 5 Hz and a sample rate of 1000 samples per second. Used as the primary input sensor for SLAM.
Wheel encoders	Two E5 series rotational encoders mounted on each of the two front wheels that turns the tracks of the platform. The encoder contains a circular disk with 500 marks and can thus theoretically detect movements of 1.2 mm or more for wheels of diameter 0.18 m. Used to estimate the distance that each track has travelled, which is utilized in dead reckoning and SLAM.
Camera	A forward-facing Raspberry Pi Camera Module v2 that is mounted on Balrog and connected directly to *RPI*. It is capable of capturing video with a resolution up to 1080p and is used to capture live video that is streamed over WiFi to BaseStation.
Ultrasonic rangefinders	Four SRF10 sensors that are mounted in each corner of the robot platform. The sensors have a maximum detection distance of 11.29 m and communicate with *Arduino-Ext* via I^2C. Could potentially be utilized in e.g. a backup obstacle avoidance system, but has no specified use in the original system design.

IMU	An MTi 100 Inertial Measurement Unit mounted at the midpoint of the robot platform. It communicates with *RPI* via USB and operates at a frequency of 50 Hz. It contains accelerometers, gyroscopes and magnetometers for estimation of the platform acceleration and angular velocity in all three dimensions. Could potentially be utilized to improve dead reckoning and SLAM, but was not found particularly useful in the previous year's project and has thus no specified use in the original system design.

3.2 SLAM

The SLAM software module runs entirely on *Laptop* and is responsible for estimating the current Balrog position and creating a map of its surrounding area, including any obstacles. It is also responsible for keeping track of which parts of the map that have been visited by Balrog. To do this, it utilizes the 360° scanning LIDAR that is connected to *Laptop*, and the wheel encoder readings that is transmitted to *Laptop* from *RPI*. The module consists of five different ROS nodes:

- *sweep_ node*
- *slam_ odom*
- *mapper*
- *slam_pose*
- *slam_ visited*

sweep_ node and *mapper* are members of external ROS packages and will not be modified in the project, whereas *slam_ odom*, *slam_pose* and *slam_ visited* will be implemented from scratch.

mapper is a ROS implementation of the OpenKarto SLAM system available in the nav2d ROS package [2]. There are however many different SLAM implementations available, including the system called GMapping which was intended to be employed in the original design. GMapping is thus described in section 3.2.1 before an overview of OpenKarto is provided in Section 3.2.2, together with a comparison of the two systems and motivation for why OpenKarto ultimately was found to be the preferred option. Each of the five ROS nodes listed above are then described in detail in the remaining subsections.

3.2.1 Background on Filter-based SLAM & GMapping

For 2D LIDAR and odometry based SLAM, the most common approach is to use a filtering based system. One such SLAM system for which there exists a highly popular open source implementation is GMapping, which was presented by Grisetti et al. in [3]. GMapping utilizes a Rao-Blackwellized particle filter and it is a refinement of the algorithm called FastSLAM that was introduced by Montemerlo et al. in [4].

In the FastSLAM approach, a mobile robot with current pose x_k is moving through an environment taking relative measurements z_k of a number of landmarks with unknown positions $m_1, ..., m_M$. The robot is driven from state x_{k-1} to x_k with the control input u_k. The SLAM problem is then to find

$$p(x_{0:k}, m | z_{0:k}, u_{0:k}), \tag{1}$$

the joint posterior of all landmark positions (i.e., the map) and the robot trajectory, given all observations and control inputs up to the current time step.

The key observation, which was originally made by Murphy in [5], is that the SLAM problem exhibits important conditional independences. Specifically, given the robot trajectory $x_0, ..., x_k$, the individual landmark positions m_j become conditionally independent. Therefore, the joint posterior of all landmarks and the robot trajectory can be factorized according to:

$$\begin{aligned} p(x_{0:k}, m|z_{0:k}, u_{0:k}) &= p(x_{0:k}|z_{0:k}, u_{0:k})p(m|x_{0:k}, z_{0:k}) \\ &= p(x_{0:k}|z_{0:k}, u_{0:k}) \prod_{j=1}^{M} p(m_j|x_{0:k}, z_{0:k}). \end{aligned} \tag{2}$$

Thus, the SLAM problem can be decomposed into a robot localization problem and M independent landmark estimation problems conditioned on the trajectory estimate. For the localization problem, FastSLAM uses a particle filter with N particles, each of which representing a trajectory hypothesis. Each particle possesses M EKFs estimating all landmark positions conditioned on this trajectory. Specifically, the joint posterior $p(x_{0:k}, m|z_{0:k}, u_{0:k})$ is represented by the sample set

$$X_k = \{X_k^{(i)}\}_{i=1}^N = \{w_k^{(i)}, x_{0:k}^{(i)}, \mu_1^{(i)}, \Sigma_1^{(i)}, ..., \mu_M^{(i)}, \Sigma_M^{(i)}\}_{i=1}^N, \tag{3}$$

where $\mu_j^{(i)}$ and $\Sigma_j^{(i)}$ gives the current Gaussian estimate of the jth landmark position conditioned on the trajectory $x_{0:k}^{(i)}$, and $w_k^{(i)}$ is the resampling weight. The FastSLAM algorithm can then be summarized by the following four steps, starting from the previous sample set X_{k-1}:

1. *Sampling*: For each particle, compute a proposal distribution π and draw a sample from it:
$$x_k^{(i)} \sim \pi(x_k|x_{0:k-1}^{(i)}, z_{0:k}, u_k) = p(x_k|x_{k-1}^{(i)}, u_k). \tag{4}$$
2. *Importance weighting*: Compute the resampling weight for each particle according to the importance function:
$$w_k^{(i)} = w_{k-1}^{(i)} \frac{p(z_k|x_{0:k-1}^{(i)}, z_{0:k-1})p(x_k^{(i)}|x_{k-1}^{(i)}, u_k)}{\pi(x_k^{(i)}|x_{0:k-1}^{(i)}, z_{0:k}, u_k)}. \tag{5}$$
3. *Resampling*: Create a new particle set by drawing particles, with replacement, from the set $\{w_k^{(i)}, x_{0:k}^{(i)}, \mu_1^{(i)}, \Sigma_1^{(i)}, ..., \mu_M^{(i)}, \Sigma_M^{(i)}\}_{i=1}^N$, with probability proportional to $w_k^{(i)}$. Give all particles in the new set uniform weight, $w_k^{(i)} = \frac{1}{N}$.
4. *Map estimation*: For each particle, perform an EKF update on the observed landmarks in order to update all $\mu_j^{(i)}$ and $\Sigma_j^{(i)}$.

In a setting where the robot is equipped with a scanning LIDAR sensor, this algorithm can also be modified to directly estimate an occupancy grid map for each particle in step 4 above, see e.g. chapters 9 and 13.10 in [6]. One then divides the surrounding area into fixed-sized grid cells and assigns each cell with a probability value describing how likely it is to be occupied, i.e., to contain an obstacle. To do this, one essentially traces each LIDAR ray and notes which grid cell the ray hit, and which cells it passed over. The hit grid cell will then be deemed more likely to be occupied, whereas the passed over cells will be deemed less likely.

The GMapping system improves the performance of this modified version of FastSLAM by two main modifications. Firstly, they draw samples from a much more accurate proposal distribution π by also taking into account the most recent observation z_k, reducing the number of required particles. Secondly, they apply a selective resampling strategy based on the effective sample size, that is they only perform the resampling step when it is really needed. This reduces the number of unnecessary resamples and thus also the risk of particle depletion.

3.2.2 Pose-graph SLAM & OpenKarto

GMapping is however far from the only available open source SLAM implementation, and a system offering very similar functionality is the OpenKarto SLAM system. Whereas GMapping is an example of a filtering based SLAM approach, OpenKarto is instead an example of an optimization based SLAM approach usually referred to as pose-graph SLAM [7].

In this approach, one extracts relative spatial constraints between different robot poses from the available sensor data, treats the history of poses as variables of an optimization problem and then tries to find the pose history that best fits all pose constraints. Relative pose constraints are extracted from odometry measurements between consecutive poses, and from matching pairs of range scans. With an estimate of the pose trajectory, one can build an occupancy grid map of the environment by projecting each range scan into the world frame.

One key concept in this approach is that of loop closures. A loop closure is when a mobile robot is able to detect that it has re-entered a previously visited area, enabling the creation of relative pose constraints between temporally separated poses. This in turn can greatly improve the mapping quality, for instance connecting the endpoints of a circular corridor.

More formally, the approach corresponds to creating a graph consisting of a set of nodes and a set of edges between pairs of nodes. Each node corresponds to a robot pose at which a range scan was taken, whereas each edge corresponds to a derived measurement of the relative pose between the pair of corresponding robot poses. A minimization problem over the robot poses is then defined from the graph by constructing an objective function where each relative pose measurement is translated into an error term.

One usually decouples this problem into two separate tasks: creating the graph from the available sensor data (graph construction), and determining the most likely configuration of the nodes given the edges of the graph (graph optimization). The system dealing with graph construction is called the SLAM front-end and its design depends on the specific set of sensors being used. The optimization system on the other hand is referred to as the SLAM back-end and deals with a general data representation which is sensor agnostic.

For its SLAM front-end, OpenKarto utilizes the work presented by Olson in [8], which is a method for matching 2D LIDAR scans using a probabilistic framework and cross-correlation.

To efficiently perform the graph optimization in the SLAM back-end, the most common method is to use techniques for nonlinear least-squares optimization, such as Gauss-Newton and Levenberg-Marquardt. These methods consist in iteratively improving an initial guess by solving a large linear system obtained from linearization of the original objective function. An open source implementation of this approach is presented by Konolige et al. in [9], which constructs the linearized system in an efficient way and utilizes sparse Cholesky decomposition to compute its solution. This implementation is utilized in OpenKarto's back-end system.

It is not obvious which of GMapping and OpenKarto that generally provides the best mapping quality. For our specific use case OpenKarto does however have one key advantage, which is that it explicitly outputs an estimate of the entire pose history and not only the current robot pose. This enables us to keep track of which parts of the map that have been visited by Balrog, even if the map were to change significantly during operation. Thus, OpenKarto will be the SLAM system employed in our proposed design.

3.2.3 sweep_node

This node is a part of the external ROS package sweep-ros, which is officially supported by the LIDAR manufacturer [10]. The node reads data from the LIDAR and publishes messages of type *sensor_msgs:LaserScan* on the topic */scan*. Each published message represents one complete 360° scan of the sensor and contains both the raw range measurements and various metadata. A schematic overview of the node is seen in Figure 3.

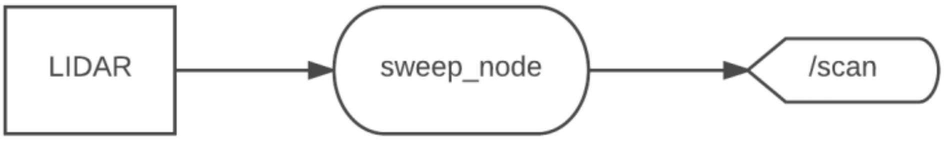


Figure 3: Schematic overview of *sweep_node*.

3.2.4 slam_odom

The node reads messages of type *std_msgs:Float64MultiArray* from the topic */rpi_data*. Each read message contains the most recent measurements of all sensors connected to *RPI*, including the wheel encoders. The node will extract the wheel encoder readings and transform these into the odometry measurement format expected by the SLAM node. The encoder data from the RPI is sent out as the angle that each wheel powering the tracks has rotated since the last measurement, $\Delta\theta$, with index left and right for each track. This can be changed in the Matlab-code to other formats, however a transformation from change in angle of the wheels to pose of Balrog is simple.

Many different approaches to transform encoder readings into a robot pose exist. The chosen method is a compromise on accuracy and complexity. Only the encoder data is used since these sensors are proven in previous projects to be quite accurate. The chosen algorithm as well as a more accurate one has been evaluated [11]. With a sampling time of 0.02 seconds for sensor data, the chosen methods present good results and are easier to implement.

Equation (6) – (7) describes how to get the change in distance, Δs for each track, given the wheel radius r_w. The model then estimates the change in angle and distance traveled, Equation (8) – (9). b is the track width, *i.e.* the distance between the tracks. Using Equation (10) – (12), a translation to change in global coordinates and angle is achieved and given the previous pose, the current pose is estimated.

$$\Delta s_{left} = \Delta\theta_{left} \cdot r_w \tag{6}$$

$$\Delta s_{right} = \Delta\theta_{right} \cdot r_w \tag{7}$$

$$\Delta\theta = \frac{\Delta s_{left} - \Delta s_{right}}{b} \tag{8}$$

$$\Delta s = \frac{\Delta s_{left} + \Delta s_{right}}{2} \tag{9}$$

$$x_{t+1} = x_t + \Delta s \cos(\theta + \frac{\Delta\theta}{2}) \tag{10}$$

$$y_{t+1} = y_t + \Delta s \sin(\theta + \frac{\Delta\theta}{2}) \tag{11}$$

$$\theta_{t+1} = \theta_t + \Delta\theta \tag{12}$$

The estimated pose is finally published as a message of type *tf:tfMessage* on the topic */tf*. A schematic overview of the node is seen in Figure 4.

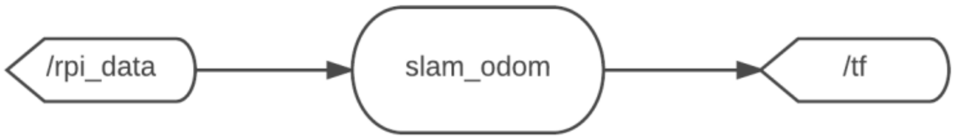


Figure 4: Schematic overview of *slam_ odom*.

3.2.5 mapper

This node is a ROS implementation of OpenKarto and takes as input LIDAR range scans read from the */scan* topic and odometry measurements read from */tf*. It outputs an estimate of the current robot pose (x, y, θ) which it publishes as a message of type *tf:tfMessage* on the topic */tf*, sampled points along the robot's estimated pose trajectory which it publishes as a message of type *visualization_ msgs:MarkerArray* on the */Mapper/vertices_ array* topic, and a map of the robot's surrounding area in the form of an occupancy grid map. This map describes the area by dividing it into fixed-sized grid cells and assigning each cell a value $q \in [0, 100]$, where a larger value of q means the cell is deemed more likely to be occupied. For a completely unknown grid cell, a value of -1 is instead assigned. The map is published as a *nav_ msgs:OccupancyGrid* message on the topic */map*, together with various map metadata such as the map width and height. A schematic overview of the node is seen in Figure 5.

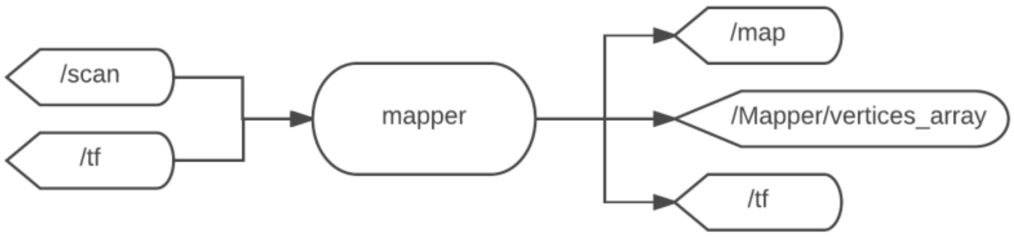


Figure 5: Schematic overview of *mapper*.

3.2.6 slam_pose

The node reads messages from the */tf* topic, extracts the estimated robot pose $(\hat{x}, \hat{y}, \hat{\theta})$ outputted by *mapper* and publishes this information as a message of type *std_msgs:Float64MultiArray* on the topic */estimated_pose*. The node functionality is not strictly necessary since the current pose estimate always will be available in implicit form on */tf*. To have this estimate available on a separate topic will however simplify the communication with external modules. A schematic overview of the node is seen in Figure 6.

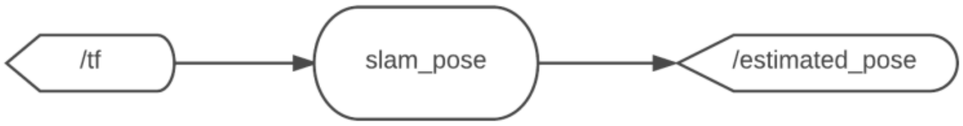


Figure 6: Schematic overview of *slam_pose*.

3.2.7 slam_visited

This node reads messages from the */map* and */Mapper/vertices_array* topics. Based on the sampled pose trajectory read from */Mapper/vertices_array* and the robot platform's known physical dimensions, it computes which grid cells that at some point have been within a certain distance of the robot. These cells are set to being visited cells by assigning a value of 1 in *map_visited*, which is a map of the same size and resolution as the one outputted by the SLAM node. All other cells in *map_visited* are assigned a value of 0. The grid cells in the map read from */map* that corresponds to visited cells are also assigned a value q close to 0, and the updated map is published on the */map* topic. The resulting *map_visited* is published as a message of type *nav_msgs:OccupancyGrid* on the */map_visited* topic. A schematic overview of the node is seen in Figure 7.

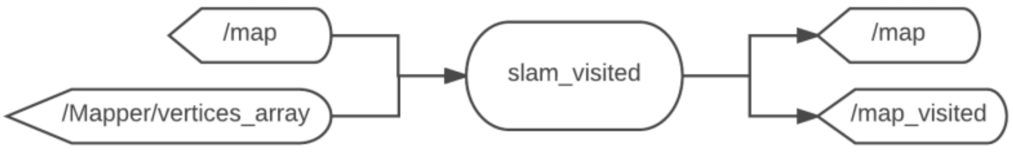


Figure 7: Schematic overview of *slam_visited*.

3.3 Navigation

The navigation module runs entirely on the *RPi* and is responsible for finding an optimized route with respect to different objectives. These objectives are:

- Mapping
- Covering tiles
- Manual exploration

Out of these three objectives mapping and covering tiles are steps in the autonomous mode. Meanwhile manual exploration requires user input. The user chooses which mode is active prior to start.

3.3.1 Autonomous Mode

The autonomous mode has been broken down into two different steps. First the Balrog will explore with mapping as main objective. Once the test area is deemed to be known the autonomous mode goes into next phase: covering tiles. By covering tiles its implied that the Balrog should cover all nodes in the test area except for nodes within the safety distance of objects. However the user shall still be able to interrupt the sequence if its desired to only execute the mapping objective. The nodes which are already covered in mapping mode are not forgotten when moved into the next phase. This holds a lot of potential for future work. One can easily imagine an algorithm that takes both the mapping and covering tiles objective into account at the same time when searching for the next goal node. However as a delimitation the group has decided not to evaluate such algorithm at the current time.

3.3.2 Manual Mode

In the manual mode the entire map is considered to be known and the user asks the Balrog to clear a path from mines between two nodes. This problem then becomes a shortest route problem between two points. The starting position which is known and the end position which is set by the user. In order to solve this problem an algorithm known as A^* will be used, which is further explained in section 3.3.6.

3.3.3 Problem Description

The mapping mode is used for getting enough information about the environment as fast as possible. It is finished when the map has information about all nodes in the area. The size of the test area is predetermined and is 5x5 meter. The starting position is known prior to start. This is an assumption which is deemed to be valid due to the fact that in a real world application a GPS would be used. The test area will be divided into a grid where the rectangular map is divided into $0.05 \cdot 0.05$ m squares. This will in turn imply that the grid will contain 10 000 nodes.

In order to avoid obstacles a safety distance of 0.5 meters is set to any known object or wall from the center of Balrog. The test area will static and no changes in the environment will be made once the Balrog is running. Each node will be evaluated and have a cost assigned to it which corresponds to how difficult it is to get to that node from the Balrogs current position. Regardless of which mode is active the Balrog shall be able to avoid any obstacles and be able handle re-routing.

3.3.4 Mapping

While the system operates in the autonomous mode and is in the mapping phase the main priority will be to discover what is previously unknown. The mapping mode will be interrupted (if only the map is desired) or automatically switched to covering each node (covering tiles mode) once a certain percentage of the map has been discovered. How large this percentage is will be determined by testing to find a suitable value. For the mapping an algorithm known as Frontier-Based Exploration will be applied [12]. The main purpose is to continuously push the boundary of what is currently known in order to gain as much new information as possible. A frontier is defined as any node which is unknown and lies next to a known node, see Figure 8.

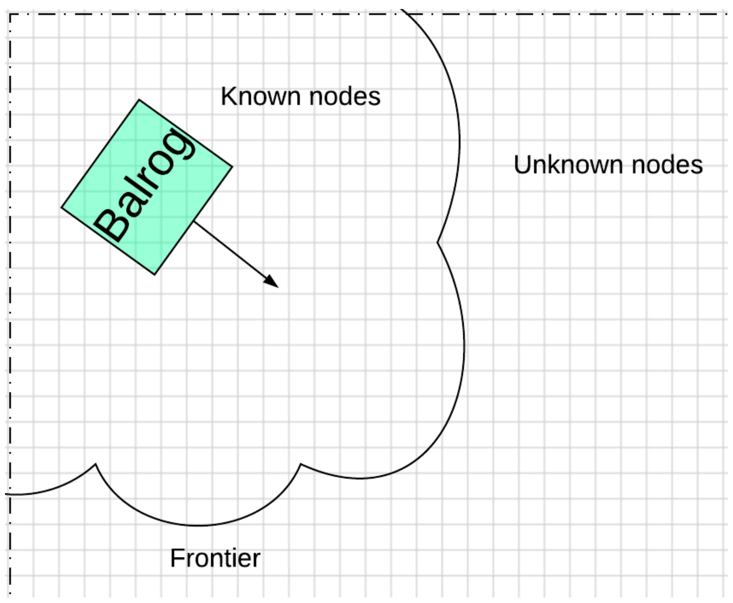


Figure 8: Schematic overview of a part of the area.

According to the Frontier-based exploration the goal node is set to be the nearest frontier which is equal to or larger than the robot itself. However since this algorithm only decides which node is set as goal node the A^* algorithm is needed in order to actually obtain a route to the chosen goal node.

The steps of the Frontier-based exploration will be:

1. Take in map information
2. End loop if the map is deemed to be complete
3. Generate all frontier nodes and place them in a list
4. Choose the frontier node closest to Balrog as the goal node
5. If the node is within safety distance from any obstacle, delete the node and go to 4.
6. Use A^* to find the cheapest path to the goal node, see Section 3.3.6
7. Feed the path outputted by A^* to the controller, one coordinate at the time.
8. Wait for Balrog to reach the frontier node.
9. Go back to 1.

3.3.5 Covering Tiles

This is basically the case when we want to find all mines in a certain area. Once the mapping is assumed to be done a new algorithm will be used in order to cover each node available. Nodes which are within 0.5 meter of any object is not included. The solution for this is still being looked into. So far there are two options for choosing a node:

1. Go to nearest node which hasn't yet been covered (use A^*).
2. Look at a downsampled area at the time and iterate to find the best path with brute force.

Both of these methods have a clear downside. For the first method the Balrog will prioritize moving straight and will not take into consideration that more nodes after the current goal node have to be covered. The second method has the potential of finding the best path possible. However due to complexity of the problem it might not be applicable with the current hardware.

3.3.6 A^* (A-star)

In order to get to a node as effectively as possible, the algorithm A^* is used [13]. The idea behind A^* is that it keeps track of how far away each node is from the starting position as well as how far away it is from the goal node. This is known as G-cost (start node distance) and H-cost (goal node distance). By summarizing these two costs (F-cost) one can weigh in the fact that it is desired to move towards the goal node by moving as little as possible from the starting node. Both these criteria have the same weight. This is done for all adjacent nodes, there are in total eight possibilities. Balrog can move as seen in Figure 9 where W stands for west, N north, S south and E east. The rotation will thereby only be made in multiples of 45°. The method is iterative and does not guarantee that an initial choice is correct. Therefore, many iterations have to be done in order to obtain the shortest path.

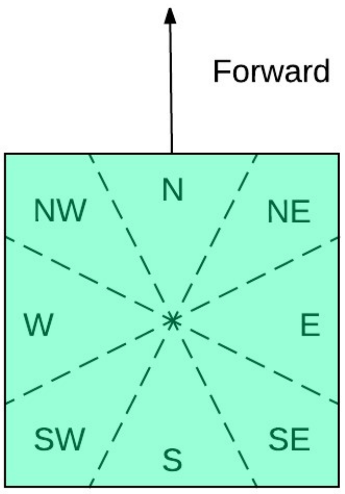


Figure 9: The directions of Balrog.

3.3.7 Cost Function

The $G-$ and $H-$ costs in the A^* algorithm will be weighted slightly differently depending on which mode is active, these are listed as shown in Table 3.

Movement \ Mode	Mapping	Exploration	Shortest path
Straight	s_M	s_E	1
Diagonal	d_M	d_E	$\sqrt{2}$

Table 3: Table of cost for moving straight and in diagonal.

s_E, s_M, d_E and d_M are cost parameters which are based on the shortest path costs but intends to punish nodes extra in certain modes. For mapping and exploring it should be avoided to run over a node more than one time. Since the odometers accuracy gets lowered when running forward and turning at the same time. The movement is limited to only one action at the time. Therefore turning takes time which is time consuming and has to be punished appropriately in each mode. If one is simply looking for the shortest distance then this factor should be neglected.

3.3.8 Code Structure

The optimization algorithm is written in Matlab and converted into C-code. The input is the pose (x, y, θ) of Balrog along with a map, Figure 10. Both *map* and *map_ visited* are received as arrays. *map_ visited* tells if a node is covered or uncovered while *map* gives the probability of a node being an object. Since the size of the map is predefined so is the length of the array. For example if the length of the test area correspond to 100 nodes which makes the whole grid contain 10 000 nodes. By starting in the origin the whole grid is built up by iterating row by row from the array. When starting mapping mode the start position will be known as well as how large the grid is. For instance the Balrog will start with its right rear edge at (0,0) which is the upper left corner of the area. However the position of obstacles and potential mines are unknown.

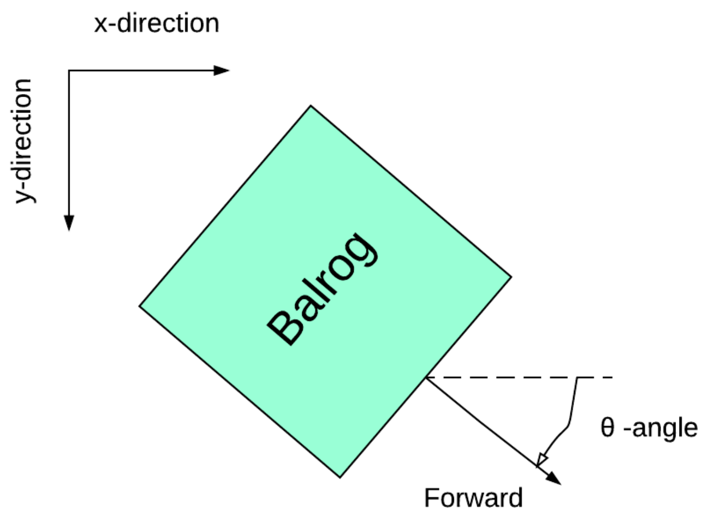


Figure 10: Coordinates for Balrog.

The output of the whole algorithm is an array of global coordinates (including the angle) of points to get to the new position, these will be given one by one to the control unit.

3.3.9 GUI

The three different strategies should be chosen on the GUI and sent via the communicator. The shortest route mode should only be able to be activated when a map already exists.

3.4 Control

The control module will initially remain unchanged compared to the previous year's implementation and will only be redesigned if found necessary. In this section we thus only provide an overview of the module functionality, a complete description is found in the Technical documentation from previous year [1].

The module receives a goal position (x_G, y_G, θ_G) from the navigation module and an estimate of the current pose (x, y, θ) from the SLAM module. Given this, it computes control signals ω_L and ω_R according to a proportional controller with a couple of extra features, where ω_L (ω_R) is the angular velocity of the left (right) platform wheel. These control signals are then sent to *Arduino-Int*.

3.5 Communication

The communication module between the *laptop* and *RPI* consists of two separate units of code: *laptop_comm* and *rpi_comm*. *laptop_comm* is a ROS node that runs on the *Laptop* whereas *rpi_comm* is a deamon running on *RPI*. *laptop_comm* and *rpi_comm* communicates via TCP over WiFi and are responsible for sending sensor data from *RPI* to *Laptop*, and sending the maps and the estimated robot pose from *Laptop* to *RPI*.

3.5.1 rpi_comm

rpi_comm runs parallel to the main program on *RPI* which contains the navigation and control modules. Both *rpi_ comm* and the main program are written in Matlab and transformed into C++ code using code generation. The sensors connected to *RPI* are read in the main program using an already implemented hardware abstraction layer (HAL). *rpi_ comm* will be based on the *Communicator* program implemented in the previous year's project, see Section 10 in the Technical documentation [1]. It will consist of two separate threads where one thread communicates with the main program (reads sensor measurements and forwards the maps and the estimated pose) using UDP and sends data to *laptop_ comm* using TCP. The other thread is responsible for opening a TCP server socket and reading data from *laptop_ comm*. The two threads communicate using a shared buffer. A schematic overview of *rpi_ comm* and its interface is seen in Figure 11.

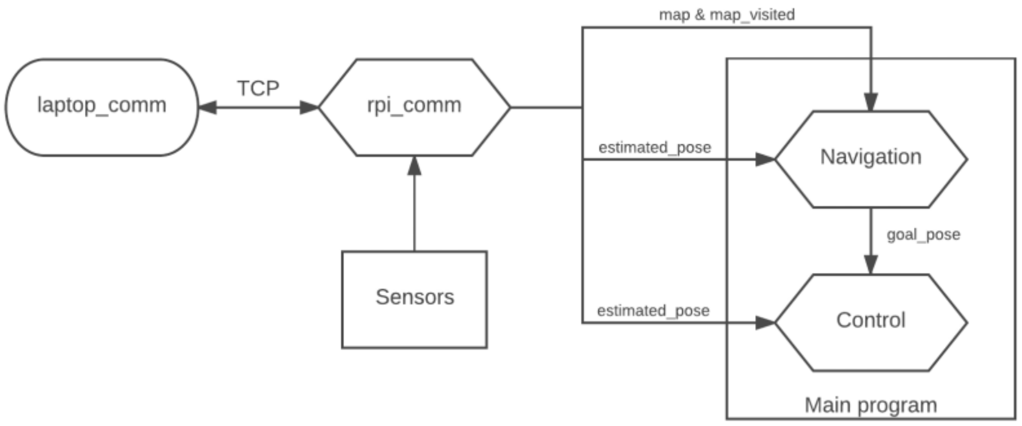


Figure 11: Schematic overview of *rpi_ comm* and its interface.

3.5.2 laptop_comm

laptop_comm is a C++ ROS node that connects to the TCP socket opened by *rpi_ comm*. It reads messages from the topics */map*, */map_ visited* and */estimated_ pose*. It transforms these messages into the protocol format and transmits the data to *rpi_ comm*. It receives the most recent measurements of all sensors connected to *RPI* from the TCP socket and publishes this information as a message of type *std_ msgs:Float64MultiArray* on the */rpi_ data* topic. A schematic overview of *laptop_ comm* and its interface is seen in Figure 12.

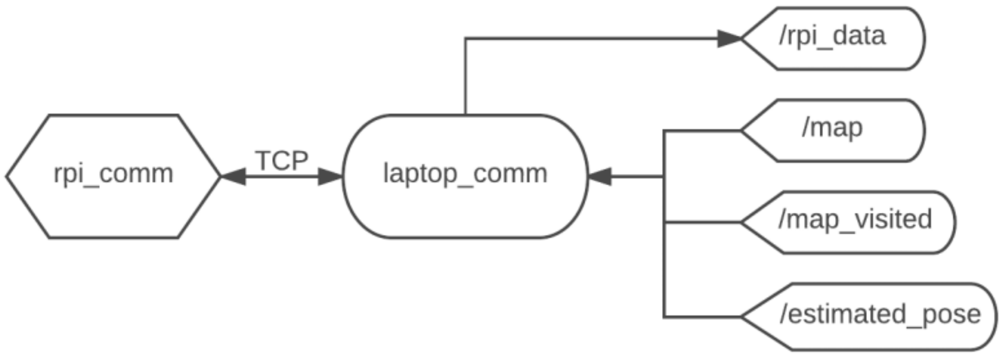


Figure 12: Schematic overview of *laptop_ comm* and its interface.

3.5.3 Protocol

All messages that will be sent between *laptop_ comm* and *rpi_ comm* begin with a message descriptor in the form of a double. All defined message types are found in Table 4 below.

Table 4: Defined message types in the communication protocol between *Laptop* and *RPI*.

Descriptor (hex)	**Descriptor (double)**	**Type**	**Content**
0x3ff0000000000000	1	Sensor data	double[4]: Ultrasonic sensors double[3]: Gyroscopes double[3]: Accelerometers double[3]: Magnetometers double[2]: Wheel encoders double[1]: Time
0x4000000000000000	2	Map	int[2]: Map size (I,J) int8[I][J]: Map
0x4008000000000000	3	Map-visited	int[2]: Map size (I,J) int8[I][J]: Map
0x4010000000000000	4	Pose	double[3]: (x, y, θ)
0x4014000000000000	5	Text	char[x]
0xffffffffffffffff	NaN	Ping	-

4 Sauron

In this section Sauron's subsystems are described in detail. A high-level hardware schematic is presented, its main hardware components are listed and implementation details are provided for its different software modules.

There are two main objectives that are prioritized within the integration of Sauron into the system.

- Sauron shall be able to estimate its relative position to a known object on the ground.
- Sauron shall be able to track a moving Balrog.

4.1 Hardware

Sauron is a commercial product that has been acquired by Saab to be integrated in to the minesweeping system. Sauron comes with the flight controller PixHawk which is an autopilot hardware running Ardupilot software.

WiFi Range Extender	Increased range for the WiFi used to transmit data to the BaseStation.

An overview of the Sauron hardware is seen in Figure 13.

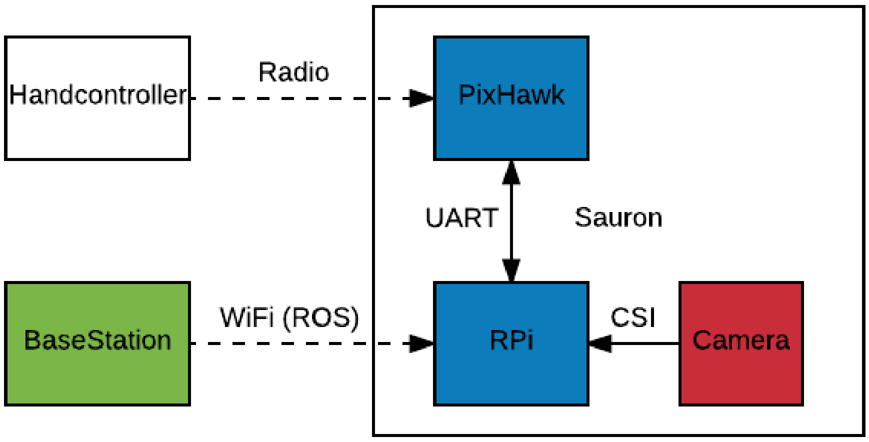


Figure 13: Schematic overview of the Sauron subsystem hardware.

4.1.1 Computing Hardware

The onboard computing hardware is listed below.

PixHawk	The PixHawk is a flight controller that runs the open source software Ardupilot. The PixHawk with Ardupilot enables advanced tools for data-logging, analysis and simulation as well as navigation and control of Sauron.
RPI	A Raspberry Pi model 3 is connected to the PixHawk onboard Sauron to enable more advanced missions such as tracking of a known object using computer vision. The Raspberry is running Ubuntu Mate Xenial with ROS (Robot Operating System) as operating system and will communicate with the ground station using WiFi.

4.1.2 Sensors

Sauron is equipped with several integrated sensors for flight control. These include an IMU, GPS, barometer and a compass. These sensors are used to control the aircraft during flight, but may also be accessed through Ardupilot if necessary. Sauron will be equipped with a Raspberry Pi Camera module V2 aimed downwards, used for observing the Balrog and other objects on the ground.

4.2 Localization

The localization module is responsible for detecting an object as well as estimating Saurons position relative the known object.

4.2.1 Detection & Positioning

To detect objects on the ground, AprilTags will be used. AprilTags are a simplified version of QR-codes. When using existing software available, AprilTags can be detected from a video stream. The Apriltag detection software will calculate the correct 3D position, orientation and identity of the tag relative the camera. The AprilTag family named 36h11 will be used.

4.2.2 Interface

The Raspberry camera onboard Sauron will capture video at a chosen framerate and resolution. The Raspberry will capture these images and transmit images to a ROS topic for compressed images (JPEG). Another topic on the Raspberry will intercept these images and run them through an AprilTag detection algorithm. The outcome of this detection is both a topic for video with overlay for AprilTags detected and a topic for information about the relative position and orientation to the detected AprilTag.

The video stream with AprilTag overlay will be recieved by the BaseStation through WiFi and will be displayed in the GUI.

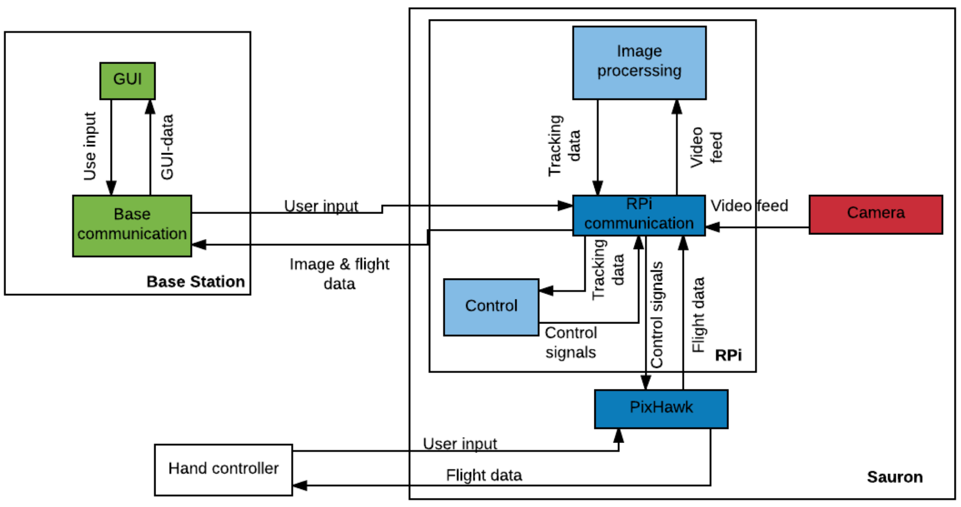


Figure 14: Schematic overview of the communication between the subsystems of Sauron.

4.2.3 Data Manipulation

The camera feed from the Raspberry camera onboard Sauron will be compressed to a stream of JPEG images. The quality of the compression will be chosen for a good compromise between image quality needed for AprilTag detection and framerate needed for automatic control of Sauron's movement.

4.3 Track Moving Balrog

The other main task is to be able to track the moving Balrog. This will be done by attaching an AprilTag to the top of Balrog that can be observed by the Raspberry Camera onboard Sauron. This will enable the system to extract information of its position relative to Balrog. By continuously updating the image data, tracking will be implemented in discrete time using controllers.

4.3.1 Detection

The detection algorithm for detecting AprilTags has already been implemented at this stage. Using this detection algorithm, position and orientation to the detected AprilTag will be obtained from the AprilTag detection node running on the Raspberry.

4.3.2 Interface

The video stream with AprilTag detection as previously implemented will be sent to the BaseStation GUI. The AprilTag detection algorithm will output relative position and orientation of Sauron compared to the AprilTag into the topic /apriltags/detections. This information will be used in a controller to minimize the deviation of position from Balrog. The interface of the system can be seen in Figure 14.

4.3.3 Data Manipulation

When detection of Balrog (AprilTag) is made, a position estimate of Sauron relative to Balrog will be available. Tracking of Balrog is done by keeping this relative position equal to a desired relative position (0 in X and Y). This is achieved by implementing a controller on the Raspberry that will minimize the position error by sending control signals to PixHawk that uses Ardupilot to navigate accordingly.

4.3.4 Controller

The altitude of Sauron in assumed to be kept constant by the built-in controller using the integrated barometer. This is the first step of implementing a controller for tracking an AprilTag. If there is sufficient time, altitude control will be implemented as well. The controller should not be dependent on GPS-fix. Therefore, the controller will not use the GPS for positioning/tracking. Furthermore, the airspace is assumed free of obstacles.

The detection algorithm outputs relative translation offsets in X and Y, called Δx and Δy, in the local coordinate system of the AprilTag. The rotational offset between ”north” according to the AprilTag and ”north” according to Sauron, φ, is also obtained. Three PID regulators will be implemented to control these offsets.

The controller for the offset in angle, φ, will be controlled independently from the other controllers. A PID regulator will be implemented to minimize the angle offset, so that the AprilTag and Sauron will align in rotation. This is done by sending commands to control the *yaw*-angle through Mavros.

The position of Sauron will be controlled using two PID controllers. These controllers relate to the angle offset φ through a rotational matrix. This matrix is customized to accommodate the chosen direction of offsets and angles according to Figure 15.

Controller for *roll* and *pitch*-angle with output to Sauron called u_α and u_β. Reference for *roll* and *pitch*-angle is *zero*.

$$\begin{pmatrix} u_\alpha \\ u_\beta \end{pmatrix} = \begin{pmatrix} K_{P(\alpha)} + K_{D(\alpha)}s + \frac{K_I(\alpha)}{s} & 0 \\ 0 & K_{P(\beta)} + K_{D(\beta)}s + \frac{K_{I(\beta)}}{s} \end{pmatrix} \begin{pmatrix} -cos(\varphi) & sin(\varphi) \\ -sin(\varphi) & -cos(\varphi) \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} \tag{13}$$

Controller for *yaw*-angle with output to controller called u_φ. Reference for *yaw*-angle is *zero*.

$$u_\varphi = \varphi(K_{P(\varphi)} + K_{D(\varphi)}s + \frac{K_{I(\varphi)}}{s}) \tag{14}$$

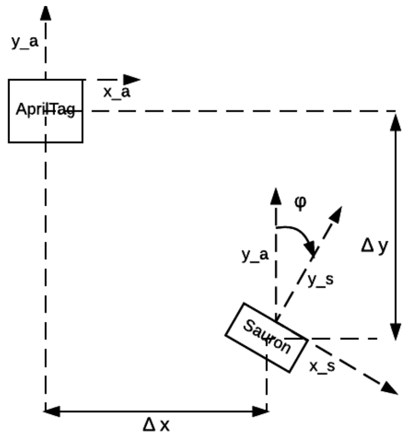


Figure 15: The chosen directions of offsets and angles. In this instance, Sauron is located in positive Δx and negative Δy. The rotational offset φ is here defined as positive.

By applying a rotational matrix, the local coordinate system of the AprilTag and the local coordinate system of Sauron can be related through the rotational offset φ. As the controller for φ acts on the relative rotation between the two coordinate systems, the control signals for *roll* and *pitch* will change according to this matrix. A positive value of α and β, positive *roll* and positive *pitch*, is defined as Sauron moving along its positive x-axis respectively along its positive y-axis in its local coordinate system. A positive value of φ, positive *yaw*, is defined as Sauron rotating clockwise around its z-axis assuming a right-oriented local coordinate system, see Figure 15. This results in that Sauron will move from its original position in a straight line (theoretically) to the AprilTag.

Sauron's four arms are not of equal length and the rotational inertia is not equal in *roll* and *pitch*-axis. Therefore, the parameters for the *roll* and *pitch* controllers have to be chosen differently from each other. Sauron needs to react equally in *roll* and *pitch* given the same input. If not, Sauron will not move in a straight line to the AprilTag given an initial offset in X and/or Y.

This method enables independent control of *roll*, *pitch* and *yaw* angles. One advantage of this is that the tracking behaviour of Sauron can easily be adjusted and tuned to handle the specific movement behaviour of Balrog. For example, Balrog likes to rotate rapidly whilst standing still. The controller for the *yaw* angle can be tuned to be fast and responsive so that Sauron reacts fairly quickly to Balrog's rotation.

If there is sufficient time, a controller for keeping the relative height to Balrog will be implemented as well. As of now, Sauron is able to maintain its absolute altitude, but will not change altitude if, for example, Balrog climbs a hill. The AprilTag algorithm outputs offset in Z, called Δz. By setting a constant reference in $\Delta z = 3m$ as defined in the requirements specification, the controller will adjust the throttle to achieve this. A PID controller is used.

$$u_T = (3 - \Delta z)\big(K_{P(T)} + K_{D(T)}s + \frac{K_{I(T)}}{s}\big) \tag{15}$$

In contrary to the control of *roll*, *pitch* and *yaw* angles, the throttle position is not *zero* when hovering, but somewhere in the middle of its range. It is therefore important that integration of the controller for the throttle is initialized at a predefined value. If not, Sauron will lose altitude when tracking mode is set by the user. Since the controller will have integral action, the altitude loss will be accounted for and compensated for so that Sauron regain an altitude of $3m$.

The frame rate, resolution and the AprilTag detection algorithm will have to be tuned so that the update frequency of the controllers can be maximized, without losing accuracy in detection of AprilTags.

The controllers will be implemented in C++. When implementing the controllers, Equation 13 - 15 will be discretized and necessary filtering of control error signals will be made. An example of necessary filtering would be a low-pass filter if the derivative part of the PID controller is implemented. The controllers will then be integrated in ROS through a Node. The input of the Node will be read from the topic /apriltags/detections and the Node will communicate to Mavros. Mavros will send the requested angles to ArduPilot which will execute required operations to perform requested tasks.

4.4 Overall Requirements

The design of integration of Sauron shall fulfill some fundamental requirements as specified in the requirements specification.

4.4.1 Safety

The user should always be able to perform a safe landing and take control over Sauron independent of what operating mode Sauron is in. A switch is already implemented on the RC controller, it sets Sauron to land at the current position when switched. The implementation will be done so that this switch will still be able to override the tracking mode and perform a safe landing.

The controller has different operating modes, besides the safety landing switch. Therefore, to take over manual control of Sauron the user only needs to change operating mode which is done by flipping another switch on the RC controller. This is all explained in the manual for Sauron.

If the connection to the RC controller is lost Sauron will land at the current location. This functionality is already implemented in the Ardupilot software and will not be interfered with.

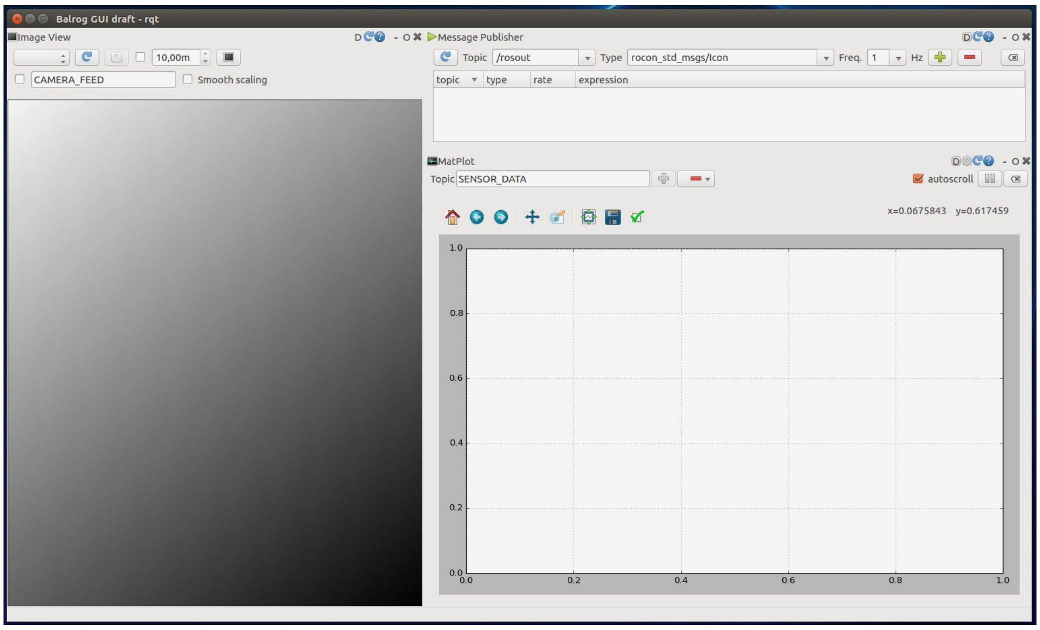
5 BaseStation

BaseStation is a computer running ROS which communicates with the Balrog and Sauron subsystems by publishing and subscribing to different topics, and displays relevant sensor data in separate GUIs.

5.1 Balrog GUI

The GUI will be implemented in the form of an rqt dashboard. It will contain rviz displaying the map and a video feed, and will allow the user to display plots of all sensors. The GUI will also allow for some input to Balrog which will be done by publishing a string containing the intended test algorithm to a ROS topic, as well as potential goal pose coordinates.

The sensor data plot will plot data from either */map*, */scan* or */rpi_ data* depending on what the user selects to be displayed. An early draft of this GUI can be seen in Figure 16.



10,00m

CAMERA_FEED

Topic /rosout Type rocon_std_msgs/Icon Freq. 1 Hz

Topic SENSOR_DATA

x=0.0675843 y=0.617459

Figure 16: Balrog GUI

The GUI module also contains a ROS node that reads the video stream transmitted from *RPI* and publishes this on a ROS topic that the GUI can display.

5.2 Sauron GUI

The GUI for Sauron will be implemented in ROS using the rqt dashboard and its functionality will be independent from the Balrog GUI and its functionality. The dashboard will include a live video feed with AprilTag detection overlay. As an AprilTag is placed on top of the Balrog, the camera feed with AprilTag detection overlay will highlight Balrog.

If there is sufficient time, the user should be able to send commands and/or switch operating modes of the flight controller of the Sauron using the GUI. The GUI for Sauron will look similar to the GUi for Balrog for which an early draft is represented in Figure 16.

6 Communication

BaseStation communicates with Balrog and Sauron by publishing and subscribing to different ROS topics. Communication between the subsystem of Balrog and Sauron is not implemented in the original system design, but could easily be implemented by adding additional topics. All currently defined topics are listed in Table 5 together with its message type, all publishing and subscribing ROS nodes, and a description of the message content. The topics for AprilTag detection is dependent on which packages, nodes, that are used. A lot more testing and research must be made until package can be chosen and topics determined.

Table 5: All currently defined ROS topics.

Message type	**Topic**	**Publishers**	**Subscribers**	**Content**
sensor_msgs: LaserScan	/scan	sweep_node	mapper balrog_gui	Single scan read from the 360° LIDAR.
std_msgs: Float64MultiArray	/rpi_data	laptop_comm	slam_odom balrog_gui	Measurements of all sensors connected to *RPI*.
tf: tfMessage	/tf	slam_odom, mapper	mapper slam_pose	Transforms between all frames related to SLAM.
nav_msgs: OccupancyGrid	/map	mapper slam_visited	laptop_comm balrog_gui slam_visited	Occupancy map outputted by the SLAM module.
visualization_msgs: MarkerArray	/Mapper/vertices _array	mapper	slam_visited balrog_gui	Sampled points on the estimated pose trajectory.
std_msgs: Float64MultiArray	/estimated_pose	slam_pose	laptop_comm balrog_gui slam_visited	Estimated robot pose outputted by the SLAM module.
nav_msgs: OccupancyGrid	/map_visited	slam_visited	laptop_comm balrog_gui	Map of all areas that have been visited by Balrog.
mavros_msgs: mavlink	/to	controler_out	ardupilot sauron_gui	Information sent to Ardupilot.
mavros_msgs: mavlink	/from	ardupilot	sauro_gui	Information sent from Ardupilot.

References

[1] CDIO 2016 TIGER. Technical documentation, 2016.

[2] Sebastian Kasperski. nav2d_karto - ros wiki, 2017. URL `http://wiki.ros.org/nav2d_karto`.

[3] C. Stachniss G. Grisetti and W. Burgard. Improved techniques for grid mapping with rao- blackwellized particle filters. *IEEE Transactions on Robotics 23(1)*, pages 24–46, 2007.

[4] D. Koller M. Montemerlo, S. Thrun and B. Wegbreit. Fastslam: A factored solution to simultaneous localization and mapping. *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2002.

[5] K. Murphy. Bayesian map learning in dynamic environments. *Advances in Neural Information Processing Systems*, 1999.

[6] W.Burgard S.Thrun and D.Fox. Probabilistic robotics. *The MIT press*, 2006.

[7] H. Carrillo Y. Latif D. Scaramuzza J. Neira I. Reid C. Cadena, L. Carlone and J. J. Leonard. Past, present, and future of simultaneous localization and mapping: Towards the robust-perception age. *IEEE Transactions on Robotics, 32(6)*, pages 1309–1332, 2016.

[8] E. B. Olson. Real-time correlative scan matching. *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2009.

[9] R. Kummerle W. Burgard B. Limketkai K. Konolige, G. Grisetti and R. Vincent. Efficient sparse pose adjustment for 2d mapping. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2010.

[10] Scanse. Github - scanse/sweep-ros: Scanse sweep ros driver and node, 2017. URL `https://github.com/scanse/sweep-ros`.

[11] Turki Y. Abdalla Ziyad T. Allawi. An accurate dead reckoning method based on geometry principles for mobile robot localization. *International Journal of Computer Applications (0975 – 8887) Volume 95– No. 13*, 2014.

[12] Brian Yamauchi. A frontier-based approach for autonomous exploration. *Navy Center for Applied Research in Artificial Intelligence.*

[13] Rajiv Eranki. Pathfinding using a* (a-star), 2002. URL `http://web.mit.edu/eranki/www/tutorials/search/`.