NTNU – Trondheim
Norwegian University of
Science and Technology

Department of Marine Technology

CS Saucer - Technical Manual

Author:
Mathias Netland Solheim

June, 2022

Contents

List of Figures

List of Tables

Chapter 1

Introduction to the CS Saucer

This chapter will provide a background on the chosen experimental surface vessel, CS Saucer (CSS), and its operational environment, the Marine Cybernetics lab (MC-lab).

Initially designed by Idland [2015], in collaboration with Ph.D. candidate Andreas Reason Dahl, the CSS is a highly maneuverable drone with a symmetric, circular hull. It was designed this way so behavior would be similar in surge and sway, thus yielding quicker response and maneuverability than a conventional ship model. Modularity was also an essential consideration in the design process, so the vessel uses interchangeable top modules that allow for various payload configurations. Sharoni [2016], for example, installed an inverted pendulum, while Ueland [2016] installed a LiDAR-scanner on a separate cover. This modularity was the main draw of the Saucer, as it would make integrating a camera with the already existing lidar relatively easy.

Recently, the other ships of the cyber fleet, namely the CS Enterprise 1 (CSE1) and the CS Arctic Drillship (CSAD), had their National Instruments (NI) compactRIO embedded computers wholly replaced. The new state-of-the-art system uses a Raspberry Pi (RPi) running a Python and Robot Operating System (ROS) based control system. Idland [2015] initially implemented the control system of the CSS on a NI LabVIEW platform as well, where the embedded hardware device NI myRIO functioned as the central processing unit. Ueland [2016] and Sharoni [2016] later replaced this system with a Robot Operating Software (ROS) based solution, running on an RPi 2 as the embedded computer in conjunction with an Arduino as part of their master theses. This provided even more flexibility in development due to the accessibility of ROS-compatible hardware and software.

A final upgrade to the Saucer was conducted as part of Solheim [2022], harmonizing the system with the other vessels.

1.1 Literature

The development of CSS is a product of much research from several theses, which contain complementary information on the theory applied to the system.

Figure 1.1: The CS Saucer with the latest module installed

1.1.1 Specialization projects and master theses

- Marine Cybernetics Vessel CS Saucer: - Design, construction and control - Idland [2015]
- Marine Inverted Pendulum - Sharoni [2016]
- Marine Autonomous Exploration using a Lidar - Ueland [2016]
- Sensor Fusion between camera and lidar for C/S Saucer - Solheim [2021]
- Integration between lidar- and camera-based situational awareness and control barrier functions for an autonomous surface vessel - Solheim [2022]

1.2 Technical specification

As part of the control system upgrade performed in this thesis, several new components were installed on the vessel. Table 1.1 provides a technical specification of the current state-of-the-art hardware utilized on the vessel. It also includes software utilized during operations. Figure 1.2 illustrates the signal and power flow between each hardware component. For a more detailed review of the components, the reader is referred to either Solheim [2021] or Ueland [2016]. Components and software added as part of the control system upgrade are marked with a star ($\star$).

Technical Specification	
Software	
Operating System	Ubuntu 20.04 LTS (Server version)*
ROS distrubution	Noetic*
Hardware	
Component	**Description**
Raspberry Pi 4b*	Embedded computer for the vessel. Handles running ROS-nodes and communication between components in the system. **Processor**: Quad-core Cortex-A72 @ 1.5 GHz. **RAM**: 8 GB LPDDR4-3200 SDRAM **Power**: 5V via USB-C
Arduino Mega	Embedded circuit board responsible for transmitting appropriate PWM signals to each component. Communicates with the RPi 4b via USB. Duty cycle between 4.3 and 9.4 %.
Raspberry Pi HQ camera*	Camera module **Resolution**: 4056x3040 (12.3 Megapixels) **Framerate**: 30 fps (at highest resolution) **Sensor**: SONY IMX477 **Lens**: 6mm wide angle
RPLidar A1	Low cost 360° 2D laser scanner with adjustable rotation speed. **Effective range:** 12 m **Sampling rate:** 2000 Hz **Angular resolution:** 1 deg **Range accuracy:** $1\% < 3\,\text{m}$, $2\% \in [3, 5]\ m, 2.5\% > 5\,\text{m}$
Torpedo 800	Motor drive for the three azimuth thrusters. Can spin the propellers clockwise or counterclockwise.
Graupner Schuttel drive unit II	Servo driver to set azimuth angle. Can rotate on the interval $[-114°, 114°]$.
Traxxas LiPo	Three cell, 11.1 V lithium polymer battery. Powers all devices on the vessel. At full charge, either 640 mAh or 500 mAh, depending on the battery used, it can power the system for several hours.

Table 1.1: Technical specification for the CSS

A second computer, referred to as the operator computer, is used together with the embedded computer on the CSS. The primary purpose of the computers is to run the computationally expensive visual detection and fusion nodes. Ideally, the computer should have a GPU to power the CNN, but since the system is designed with low computational availability in mind, a powerful CPU suffices. Therefore, for most of this thesis, a Dell laptop equipped with an 8-core Intel i5 vPro processor was utilized. Like RPi, the computer ran Ubuntu 20.04 LTS and ROS Noetic.

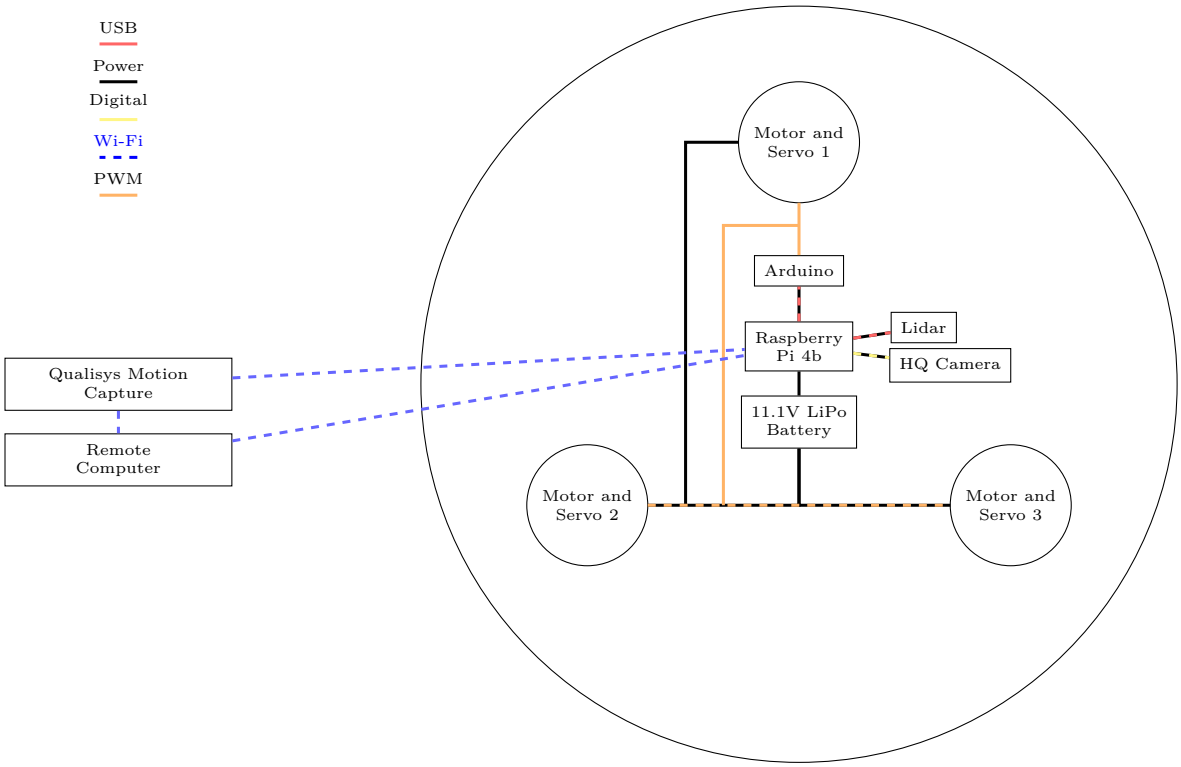

Figure 1.2: Signal and power flow between system components.

1.3 Software

1.3.1 Robot Operating System

The framework of the vessel's new control system is entirely based on the Robot Operating System (ROS). This section will, therefore, provide a brief introduction to ROS and its basic concepts.

Introduced in 2007, ROS is an open-source project that provides tools, libraries, and conventions for robot applications. It functions as a meta-operating system (OS) handling services you would expect from a conventional OS. These include hardware abstraction, message-passing between processes, and package management.

A ROS process is represented as a node in a graph architecture. Nodes are connected to edges known as topics, through which they can pass messages to one another. They can also provide and make service calls to each other and send or retrieve data from a common parameter server known as the ROS-master. The ROS-master registers all active nodes to itself and establishes the peer-to-peer communication network of the nodes. Figure 1.3 illustrates the basic communication of a ROS-system.

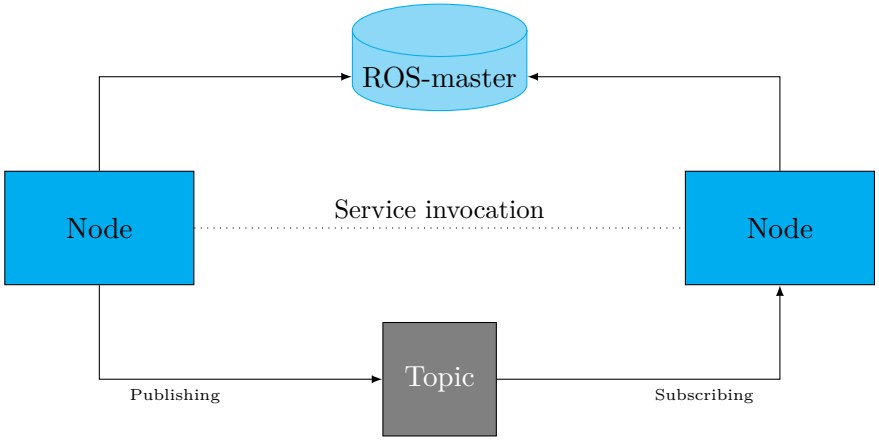

Figure 1.3: Basic ROS concept

This decentralized architecture is the main strength of ROS, as it allows nodes to be run on separate, networked hardware. As each node process is isolated and messages passed between standardized, the implementation language of the node is also irrelevant. Effectively this means that one can run one or more nodes written in C++ in conjunction with nodes written in, for example, Python. This ties in with the last strength, the ROS ecosystem. ROS offers many easy and accessible software for robots as an open-source project, making integrating sensors a simple task. Most hardware comes with ROS support from the manufacturer or a third-party individual. As mentioned before, the software language is irrelevant, so one can easily download a C++ ROS driver and run it with a primarily Python-based system.

1.3.2 ROS architecture

Figure 1.4 illustrates the different node processes and message flow in the proposed ROS vessel control system from Solheim [2022]. Nodes are depicted as circles, with orange and red being sensor-related, blue being control system modules, and blue-grey being hardware nodes. *Topics* are rectangles connected to nodes. If an arrow points from a node to the topic, the node is publishing to the relevant topic. Accordingly, arrows pointing to nodes from topics mean that the node subscribes to the given topic. The *gain server* is a parameter-server that receives control- and observer gains from the operator, allowing the user to tune the system during operation.

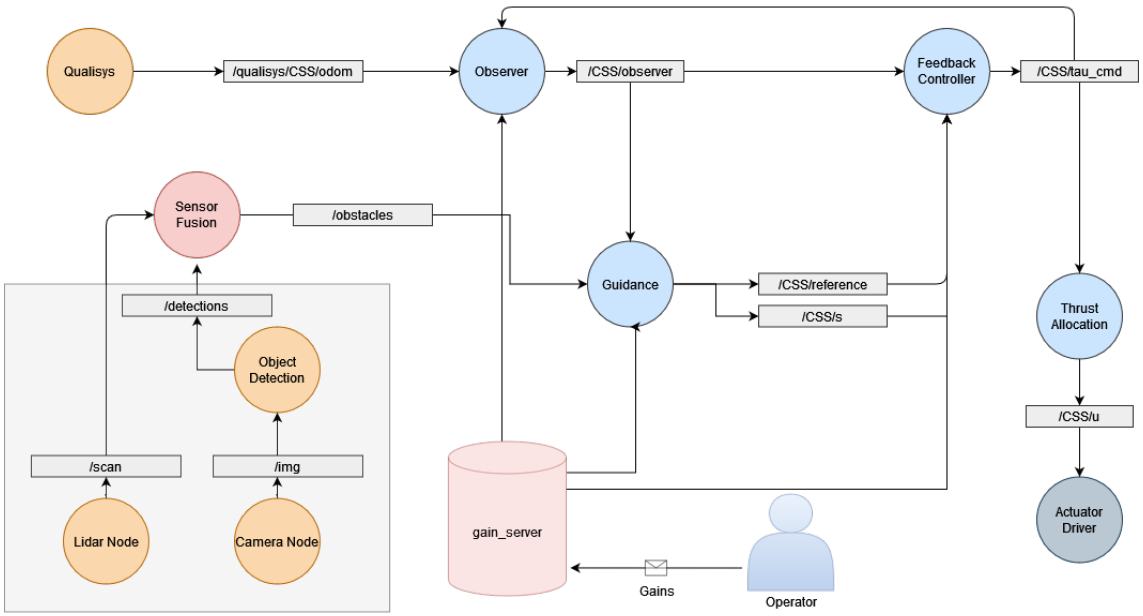

Figure 1.4: ROS-architecture of the CS Saucer.

1.4 Vessel model

A mathematical model describing vessel dynamics is desirable in designing a control system and performing simulations. Starting from the equation of motion for a vessel at sea [Fossen, 2021], one can derive the necessary equations for the control model

$$\mathbf{M}_{RB}\dot{\boldsymbol{\nu}} + \mathbf{C}_{RB}\boldsymbol{\nu} + \mathbf{M}_{A}\dot{\boldsymbol{\nu}}_r + \mathbf{C}_{A}(\boldsymbol{\nu}_r)\boldsymbol{\nu}_r + \mathbf{D}(\nu_r)\nu_r + \mathbf{D}\boldsymbol{\nu}_r + \mathbf{g}(\boldsymbol{\eta}) = \tau_{ext}, \tag{1.1}$$

where:

- $\boldsymbol{\nu} = \begin{bmatrix} u & v & r \end{bmatrix}^\top$ is the body-fixed velocities in surge, sway and yaw.
- $\boldsymbol{\nu}_r$ is the body-fixed velocities relative to local current in surge, sway and yaw
- $\mathbf{M}_{RB}$ and $\mathbf{M}_A$ is the vessel inertia matrix for mass and added mass
- $\mathbf{C}_{RB}$ and $\mathbf{C}_A(\nu_r)$ is the vessel Coriolis matrix for rigid body and added mass respectively.
- $\mathbf{D}(\nu_r)$ is the nonlinear damping matrix
- $\mathbf{D}$ is the linear damping matrix
- $\mathbf{g}(\eta)$ is the hydro-static restoring matrix
- $\boldsymbol{\tau}_{ext} = \begin{bmatrix} X & Y & N \end{bmatrix}^\top$ is the external forces acting in surge, sway and yaw

A model derivation was conducted by Ueland [2016] for his master thesis, which will be reused in this project. The following assumptions were made for the model:

1. Zero current in the MC-lab, $\nu_r = \nu$.
2. The CSS is self-stabilizing by hydrostatic forces in heave, roll, and pitch. The rotations are considered small, so movements in surge, sway, and yaw are not affected by the configuration in pitch and roll.
3. No hydrostatic restoring forces in surge, sway and yaw, $\mathbf{g}(\eta) = 0$.
4. Constant frequencies, meaning that damping and added mass are also considered constant.
5. The hull of the CSS is assumed to be completely symmetric.

The resulting control design model is equivalent to the simplified model presented by Fossen [2021], which is a good representation of a 3DOF marine craft not affected by environmental forces, that is,

$$\begin{aligned} \dot{\eta} &= \mathbf{R}(\psi)\nu \\ \mathbf{M}\dot{\nu} + (\mathbf{C} + \mathbf{D} + \mathbf{D}(\nu))\nu &= \tau, \end{aligned}$$

where:

- The inertia matrix $\mathbf{M}$

$$\mathbf{M} = \begin{bmatrix} 9.51 & 0 & 0 \\ 0 & 9.51 & 0 \\ 0 & 0 & 0.116 \end{bmatrix} \tag{1.2}$$

- The Coriolis matrix $\mathbf{C}$

$$\mathbf{C} = \begin{bmatrix} 0 & -9.51r & 0 \\ 9.51r & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{1.3}$$

- The linear damping matrix $\mathbf{D}$

$$\mathbf{D} = \begin{bmatrix} 1.96 & 0 & 0 \\ 0 & 1.96 & 0 \\ 0 & 0 & 0.168 \end{bmatrix} \tag{1.4}$$

- The non-linear damping matrix $\mathbf{D}(\nu)$

$$\mathbf{D}(\boldsymbol{\nu}) = \begin{bmatrix} 7.095|u| & 0 & 0 \\ 0 & 7.095|v| & 0 \\ 0 & 0 & 7.095|r| \end{bmatrix} \tag{1.5}$$

- The rotation matrix $R(\psi)$

$$\mathbf{R}(\psi) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.6}$$

1.5 Sensor integration

A new top module was issued for the CSS as part of Solheim [2022] On top of it, two sensors were integrated along with tracking markers for the Qualisys motion capture system present in the MC-lab. The Qualisys system acts as a GPS for the vessel, providing the control system with measurements of the vessel's position and orientation. The markers are mounted on slender rods of varying heights along the circumference of the module. Then, the lidar is mounted in the center, directly above the CO of the CSS. The sensor is oriented so the x_l and y_l axis of the lidar align with the surge and sway directions, respectively. While the lidar provides a 360-degree scan, the interval of $[-90°, 90°]$ is most interesting. Thus markers are mounted to cause as little interference as possible and still provide an accurate estimation of the vessel states. The same constraints apply to the placement of the camera. It requires an unobstructed view but can not interfere with the lidar in the critical area. Therefore it is mounted on a slender rod right behind the lidar with enough clearance to the lidar that it sees the environment clearly. The camera is tightly connected to the vessel heading, always pointing in the direction that is considered forward. Accordingly, it is mounted so that the z_c- and x_c-axis align with the surge and sway directions of the CSS.

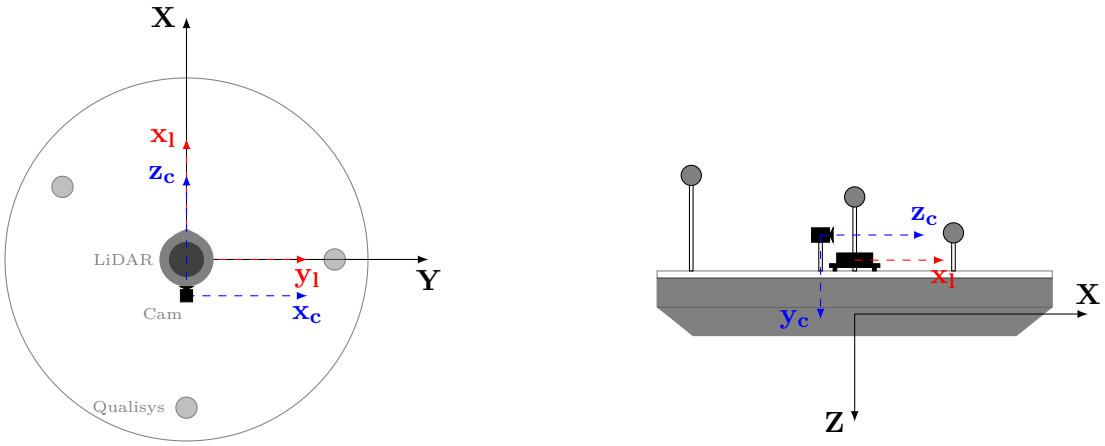

Figure 1.5: Sensor integration for the CS Saucer

The open-source computer vision library OpenCV is utilized to establish an image stream between the camera and the object detection module. It provides a generic ROS camera driver that can be utilized with all Video4Linux (V4L2) compatible cameras and publishes the image stream to the ROS-topic `/cv_camera/image_raw`. In addition, Slamtec, the manufacturer of the RPLidar, provides a ROS driver for their hardware. The driver publishes the measured ranges to the topic `/scan`.

Chapter 2

Getting started with ROS

This manual is intended to provide an introduction to the basic ROS procedures necessary to operate the CS Saucer in the Marine Cybernetics Library. It is intended for the use of future students at NTNU, whom to utilize the vessel for projects. The procedures presented in this manual are also generic enough that they apply to the other vessels of the cybership fleet.

The author of this manual has spent countless hours troubleshooting software, and setting up an environment suitable for operation on the Raspberry Pi. The hope is that the provided procedures will shorten the time future students will have to use by providing better documentation.

2.1 Raspberry Pi Image

Currently, all the vessels in the MC Lab run a Raspberry Pi 4B as their embedded computer. The CSE1 and CSE1 run the same raspberry pi image with Raspbian Buster as the OS and Melodic as their ROS-distribution.

For this thesis, the Raspbian Buster installation was insufficient, so a new image was created for this thesis. The main problem was that Raspbian Buster contained a fatal bug that prevents it from connecting to enterprise networks such as eduroam. Considering that much of the work is done on campus ground, and the Saucer required downloading a lot of software, a new operating system was installed for the Saucer. At the time of writing it runs Ubuntu 20.04 LTS Server edition, and ROS Noetic. Everything needed to run basic ros is installed, along with the computer vision library Open CV, Tensorflow and scipy as well.

The student can choose which OS suits their needs and make an image of the memory card on either the CS Saucer or CS Enterprise for their own use. More information about this can be found here. It is recommended to copy one of the memory cards rather than attempting a install from scratch.

2.2 Communicating with the Raspberry Pi

To communicate with the Raspberry Pi on the vessel, one needs to be connected to the 'MC lab' local network. After connecting to the network, open up a terminal and see if the vessel is also connected by running the command:

```
$ ping 192.168.0.108
```

Here the number-sequence is the IP of the Saucer. If the RPi responds you can SSH onto it and run files via the terminal. Simply run the command

```
$ ssh ubuntu@192.168.0.108
```

Most likely you will then be prompted for a password. All the RPi's have ***marin33*** as the designated password. You are now ready to run commands on the raspberry pi remotely.

2.3 How to ROS: A step by step guide

This section covers the basic ROS-commands you need to properly run scripts on either the RPi or your Ubuntu PC.

2.3.1 Sourcing ROS

ROS-programs are generally run using command-line tools. Thus, we have to source our installation when we open a terminal. To source the installation run the command

```
$ source /opt/ros/$ROS_DISTRIBUTION$/setup.bash
```

The ROS-distribution will either be called *melodic* or *noetic*, depending on which vessel you use. On the provided hardware ROS is sourced automatically via a bash-script every time you open the terminal, but if you are using your own computer you will have to do this for ROS commands to function.

2.3.2 Creating a workspace

Next, we need to create a designated workspace for our project. Navigate to the desired location on your computer (/documents, for example) and create a new folder for your project by running the following command

```
$ mkdir -p my_folder/src
```

This creates a folder with a sub-directory called *src*. You can replace *my_folder* with any name you want, but *src* must be kept the same. Standard ROS-convention is prefixing your workspace folder with *ws*, e.g *ws_dplab*.

Next, navigate to the workspace folder, and run the command

```
$ catkin_make
```

This will build your workspace environment with the proper ROS dependencies. After running this command, you should be left with a directory structure that looks like this:

```
my_folder
├── build
├── devel
└── src
```

You are now ready to run ROS-files. As you add more packages to your project, it is good to rebuild your project. This is done by running the *catkin_make* again.

2.3.3 The src-directory

This is the folder you will be working the most in, and were you will be putting all your ROS-packages. Each of these packages contain its own *src*-directory where the scripts you will be using are located. A typical project folder can look something like this:

```
my_folder
├── build
├── devel
└── src
    ├── simulator
    │   ├── launch
    │   └── src
    │       └── CSS.py
    └── feedback_controller
        ├── launch
        └── src
            └── ctrl_joy.py
```

2.3.4 Running ROS nodes

After your code is written it is time two activate your ROS-nodes.

2.3.4.1 The ROS-master

First we need to make sure the ROS-master is running. On the raspberry-pi, this should be activated automatically. If you are running on the Ubuntu-computer you will have to enable it manually. Open a new command-line window and run the command:

```
1 $ roscore
```

Now the ROS-master is running, and you can start activating nodes.

If you are running nodes on separate computers, you need to make sure the computer not running the master knows were to find it. This is done by exporting the url for the ROS master. Before running any nodes, use the following command:

```
1 $ export ROS_MASTER_URI=http://192.168.0.108:11311
```

It is also smart to export the ROS_IP. This tells the master were the signals are coming from. You can determine your IP by running the command `ifconfig` and then

```
$ export ROS_IP=YOUR IP
```

2.3.4.2 Activating nodes

In a separate command-line window, navigate to your project folder. Then source the setup.bash file in *devel*.

```
$ source devel/setup.bash
```

To activate a node, we first need to make sure that the script is executable. To make a file executable, navigate to the relevant directory and run the following command:

```
  $ chmod +x <node-script>.py
```

If the script has become executable, the file-name should be green the next time you run **ls** inside the directory. Then, to activate a node run navigate to the base directory of your workspace and run

```
$ rosrun <package-name> <node-script>.py
```

where <package-name> is the name of the package you want to run, e.g "controller", and the <node-script>.py is the python script that initializes the rosnode, e.g "ctrl_joy.py".

2.3.5 Launch files

As a project progresses, more and more nodes are typically added. The process of activating nodes can therefore become more tedious as the complexity increases. To avoid having to open new command-line windows for every node, we can instead use *launch-files* to activate multiple nodes simultaneously. A launch file is easy to create, and is placed in the *launch* directory of a package, as illustrated in the directory-tree in section 2.3.3.

Listing 1 Example of a launch-file

```
<launch>
    <node name="simulator"  pkg = "simulator" type="CSS.py" />
    <node name="observer"   pkg = "observer"  type="observer.py" />
    <node name="guidance"   pkg = "guidance"  type="guidance_CBF.py" />
    <node name="controller" pkg="feedback_controller"  type="cascade_backstepping.py" />
</launch>
```

Listing 1 shows the basic setup of a launch file. This example launches four separate nodes, the simulator, an observer, the guidance module and the controller module.

To run a launch file we use the following command:

```
$ roslaunch <package_name> my_launchfile.launch
```

Replace package_name with the name of the package you placed your launch file in. All launch files have the .launch ending (e.g DP-system.launch) .

2.3.6 Topics

When nodes are activated, they will start to either *publish* or *subscribe* to *topics*. Topics are named buses over which nodes exchange messages. We can any topics that are being published/subscribed to in a ROS-system by using the command:

```
1 $ rostopic list
```

This will display all the active topics in a list in your terminal. In a DP-system we might, for example, have a thrust allocation algorithm that publishes actuator commands u to a topic that a the driver of the thrusters subscribes too. This topic is called something like **CSS/u**. If we wish to display the signals from this topic in real time we can run the command

```
1 $ rostopic echo CSEI/u
```

This will print every message that is sent to the given topic, and can be a useful tool when debugging.

2.3.7 Storing data

It is desirable to store the message-signals in the system in some format so that we can later analyze and plot the data. ROS provides its own tools and file-format for this, called a *bag* file. After we have launched some nodes we run the command

```
1 $ rosbag record <topic>
```

<Topic> is replaced with the names of the topics we wish to save. Multiple topics can be recorded at the same time, in the same bag file. You just have to list the topic names with a space. If i for example wanted to record the commanded force signal from our control law and the resulting actuator commands form the thrust allication i would use

```
1 $ rosbag record CSS/u CSS/tau
```

When you have collected sufficient data, simply use CTRL+C to abort the operation. The data will be saved in a bag-file in your workspace.

2.3.8 Other usefull ROS-commands

```
1 $ rosnode list
```

Shows a list of all active ROS-nodes

Chapter 3

Control system manual

3.1 System requirements

The following is a list of all software required to run the control system implemented in this thesis. The are split into to parts, general software, python libraries and ROS-packages.

Requirements	
General	
Operating System	Ubuntu 20.04 LTS (Server version)
ROS distrubution	Noetic
Python	3.8+
GCC	8+
OpenCV	
Python libraries	
Numpy 1.19.5+	
tensorflow	
openCV	
scipy	
qp_solvers	
cvxopt	
ds4drv	
Ros packages	
CV-camera	
CV-bridge	
rplidar	
rosserial	
ds4_drivers	
dynamic_reconfigure	

Table 3.1: Required software

3.2 Running the control system nodes

This section will describe the procedure for running the available control system. First, one must SSH onto the raspberry pi as described in Section 2.2. Den navigate to the workspace. It should be called `ws_saucer`. Source the setup.bash file, and we are ready to start.

3.2.1 Dualshock 4 driver

The system relies on the playstation ds4-driver, so we start by launching this node. Its package should be in the directory. To activate the driver, run the command

```
1 $ roslaunch ds4_driver ds4_driver.launch
    use_standard_messages:=True
```

Running a roslaunch command will automatically start the ROS-master if you do not have one running. To connect the bluetooth controller, simply press the button on the middle. The controller should light blue when connected and an ouput should be printed in the terminal window. If the controller is not paired, you can follow this tutorial.

3.2.2 Camera

Next up is the camera node. Open a new terminal. For this driver you do not have to be in the workspace.

```
1 $ rosrun cv_camera cv_camera_node
```

You should see an ouput that the calibration file was loaded if successfully activated.

3.2.3 Lidar

To activate the lidar, open a new terminal and navigate to the workspace folder again. Then run the command

```
1 $ roslaunch rplidar rplidar.launch
```

Again, if successful you should see an output in the terminal. If it tells you the firmware is deprecated, just ignore the warning.

3.2.4 Arduino

Finally, before we activate the motion control system we must enable one final node. This is the node on the Arduino that subscribes to the pwm signals. To enable it, run

```
1 $ rosrun rosserial_python serial_node.py _port=:dev/ttyACM0
```

Here the final argument specifies which usb port the arduino is connected to on the raspberry pi. It may vary, so to check run

```
1 $ ls /dev
```

to see which devices are connected.

3.2.5 Motion Control System

Finally, we can activate the motion control system. In the workspace folder run

```
$ roslaunch feedback_controller collision_avoidance.launch
```

This should activate necessary nodes. The system defaults to manual control with the joystick controller, but can switch to automatic by pressing the triangle button on the ds4 controller. The guidance is activated by pressing square. Note that this should not be done unless the vessel is in the basin, and the object detection node is active.

If one wishes to for example test one node, each package can be launched individually using the guidelines from Section 2.3.4.

3.2.6 Object detection

On your operator computer, navigate to the workspace folder of your choice. Remember to export the ROS master url to establish communication between the nodes. To run the SA system use the following command

```
$ rosrun objdetection detection_node
```

This will open a window with video-feed from the saucer on your operator computer.

3.3 Dynamic Reconfigure

To make tuning your controller and observer more effective, tools for dynamic tuning are provided in the form of the ROS-package gain_server. This allows you to change the gains in real time, rather than having to hard-code the gains and then stopping and starting your system every time you want to change them. To activate the node, and GUI use the following steps

Run the node use the command:

```
$ rosrun gain_server server.py
```

NOTE! If you use the dynamic tuning, you should always activate this node **first** or you may experience errors. This is because the subscriberss in guidance, observer and controller nodes will expect values to be there when they are not. When the node is activated, the initial gains should be printed to your terminal. If you use the collision_avoidance.launch file the gain_server node will be started as part of this, and you can disregard the first step.

In another terminal, activate the GUI:

```
$ rosrun rqt_gui rqt_gui -s reconfigure
```

This should open a program that looks like this:

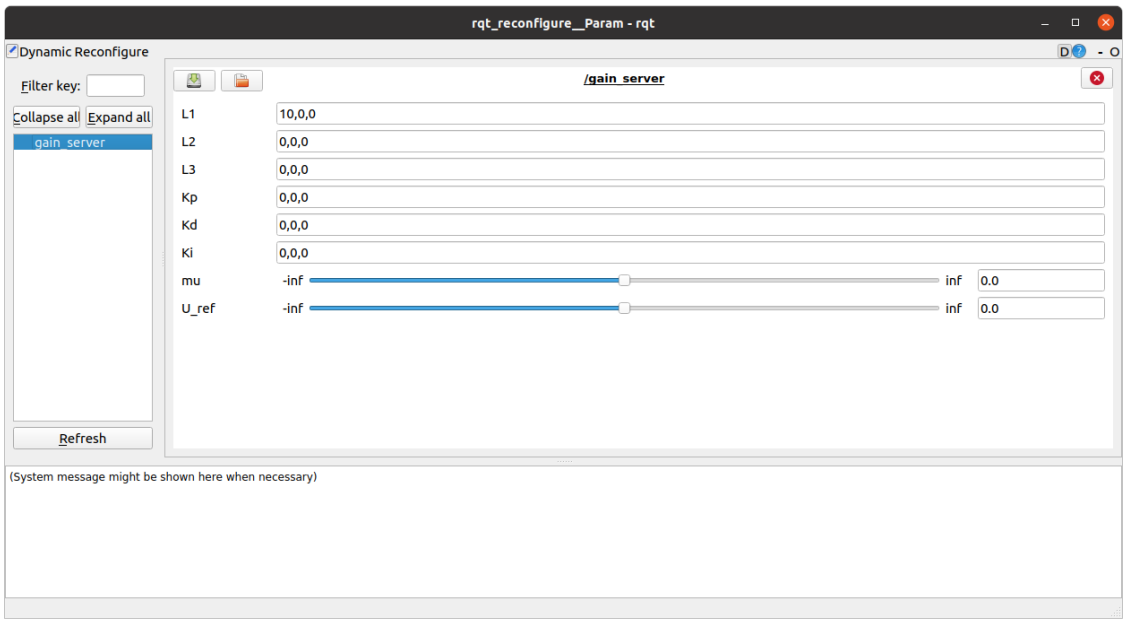

Figure 3.1: Tuning GUI

Each field or slider here corresponds to a gain in either the observer or controller. Not all the variables are relevant either. The last to are single float values, while the first six are meant to represent the elements on the diagonal of a 3×3-matrix. Change these to tune you controller.

Chapter 4

Calibration Procedures

This manual is meant to walk the user through calibrating the situational awareness system of the CS Saucer. It encompasses the camera calibration and then the lidar and camera calibration procedures.

4.1 Camera calibration

First, make sure that the camera is correctly mounted, and that the ribbon cable is connected to the Raspberry Pi. For this calibration, you will need a checkerboard pattern. You can find some in the MC Lab, or print out your own on an A3 paper. It is recommended that you laminate the checkerboard so you can use it multiple times. The pattern used in this thesis was 9x7 squares with the length of 40 mm. The exact number and size is arbitrary as long as it is specified to the software. Place the Saucer somewhere high enough that you can move quite freely in the camera frame.

SSH on to the Raspberry pi via your operator computer, and run the camera driver as outlined in Section 3.2.2. Then in a separate terminal window on the operator computer (not the RPi!) run the following comand:

```
1 $ rosrun camera_calibration cameracalibrator.py --size 8x6 --
    square 0.40 image:=/cv_camera/raw_image camera:=/cv_camera
```

Here –size is the argument for number of inner corners in the pattern and –square the length of the sides. If the node is successfully activated, you will be met with the following window:

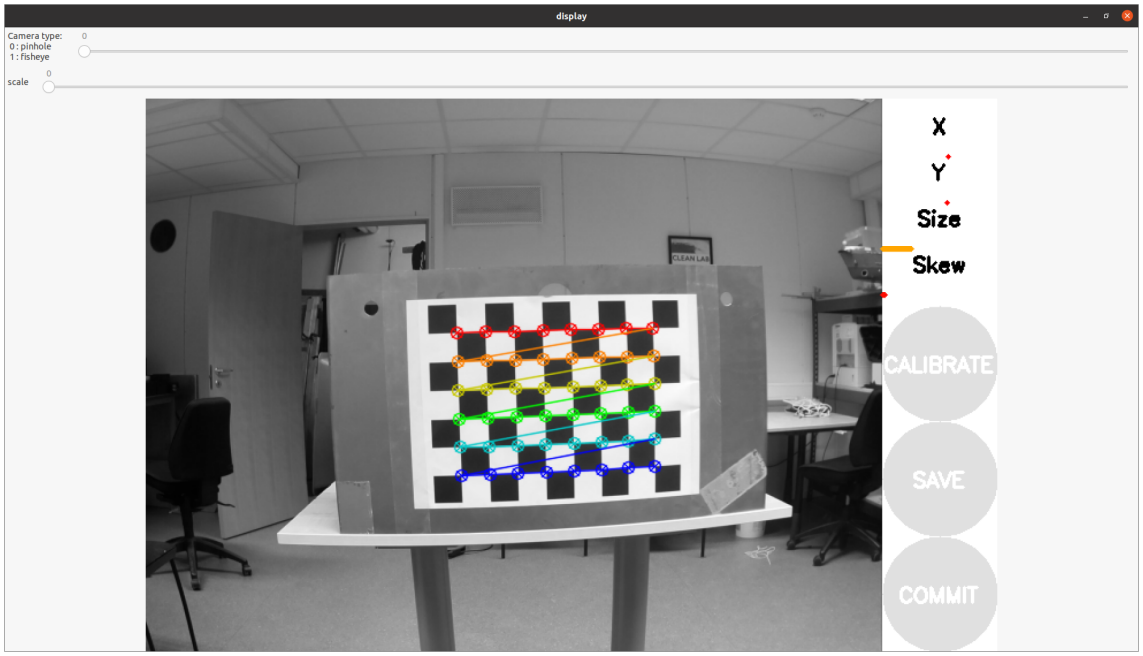

Figure 4.1: Calibration window

Move the checkerboard around in different orientations, at different distances in the camera frame. The program will grab data-samples as you go. When the bars on the right of Figure 4.1 are full enough, the calibrate button will turn a darker shade of grey. This means that enough data has been collected to produce a calibration. You can keep going to produce a more accurate calibration. When you are finished, simply press the cailbrate button and the procedure will start. After it is over, the intrinsic parameters will be printed to the terminal window. To save these, press commit. This will make sure that the camera driver on the raspberry pi, always launches with these intrinsic parameters. You can also save the parameters to a file using the 'Save' button. Now you can exit the window.

4.2 Camera-Lidar calibration

The software for the lidar-camera calibration is based on ehong-tl, 2019. However, some changes have been made, so it is recommended that you use the files provided electronically with the thesis. Copy the calibration workspace over to your operator computer, and build it. Make sure you clean the files in the directory *data*. In the directory calle *config*, edit the config.yaml file to contain the intrinsic camera parameters you obtained from the camera calibration.

Next you should identify the geometry you wish to use as your anchor points, and measure the height up to the lidar from the floor. Then, mark that height with tape on the corners inside the cameras point of view.

Then you can SSH onto the RPi and start the lidar and camera according to Section 3.2.3 and Section 3.2.2. On the operator computer, in the calibration workspace, run the command

```
1 $ roslaunch camera_2d_lidar_calibration
    collect_camera_calibration_data.launch
```

This will print you camera parameters in the terminal and activate the program RViz, as seen in Figure 4.2

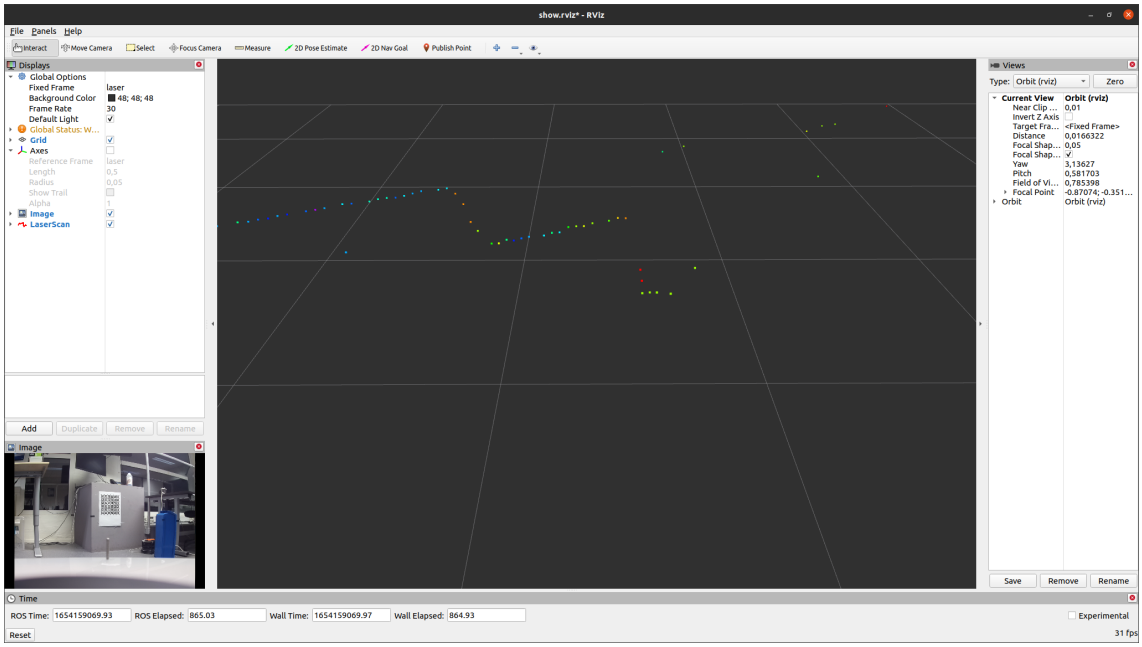

Figure 4.2: RViz window for calibration

Then using the **2D Nav Goal** tool from the top bar, pick a point from the laser point cloud that corresponds to a corner. This will prompt the following window in Figure 4.3

Figure 4.3: Choosing corresponding pixel value

Mark the area in the image that corresponds to the lidar point you chose. This is not an exact science, but try to be as accurate as you can. When satisfied with the point placement, press space and the window will close and the point be saved. Repeat the

process several times for all the corners until you get enough data points. 10-20 should suffice, try to reduce the RMSE as much as possible. After you have enough data, close RViz and in the terminal run the second command

```
1 $ roslaunch camera_2d_lidar_calibration calibration.launch
```

This will run the camera and lidar calibration procedure and spit out the rigid body transformation along with the root mean square error. You can check the translation with hand-measurements, if they are in the same ballpark the calibration is ok. You can now try to reproject the laser point onto the live image. This is done by running

```
1 $ roslaunch camera_2d_lidar_calibration reprojection.launch
```

Here you can see how the lidar points line up with the tape.

References

ehong-tl (2019). URL: https://github.com/ehong-tl/camera_2d_lidar_calibration.

Fossen, Thor I. (2021). *Handbook of Marine Craft Hydrodynamics and Motion Control*. John Wiley and Sons Ltd.

Idland, Tor Kvestad (2015). "Marine cybernetics vessel cs saucer:-design, construction and control." MA thesis. NTNU.

Sharoni, Rotem (2016). "Marine Inverted Pendulum." MA thesis. NTNU.

Solheim, Mathias (2021). "Sensor Fusion between camera and lidar for C/S Saucer, Specialization project." MA thesis. NTNU.

– (2022). *Integration between camera- and lidar-based situational awareness with control barrier functions for an autonomous surface vessel*.

Ueland, Einar Skiftestad (2016). "Marine autonomous exploration using a lidar." MA thesis. NTNU.