NTNU – Trondheim
Norwegian University of
Science and Technology

Technical Manual
Dynamic Positioning lab - TMR4243

Version 2.0.0

Mathias Solheim

September 1, 2022

Contents

1 Introduction **2**
1.1 Robot Operating Software 2

2 Hardware **3**

3 Software Prerequisites **3**

4 Networking **3**

5 How to ROS: A step by step guide **4**
5.1 Sourcing ROS 4
5.2 Creating a workspace 4
5.3 The src-directory 5
5.4 Running ROS nodes 5
5.4.1 The ROS-master 6
5.4.2 Activating nodes 6
5.5 Launch files 6
5.6 Topics 7
5.7 Storing data 7
5.8 Other usefull ROS-commands 8

6 ROS templates **8**
6.1 Node-structure 8
6.2 Common_tools package 8

7 PlotJuggler **9**

8 Dynamic Reconfigure **10**

List of Code Listings

1 Example of a launch-file 7
2 Example of a node-file 12

1 Introduction

This document provides aims to provide the necessary technical insight to utilize the software and hardware provided in the Dynamic Positioning lab of the course TMR4243 - Marine Control Systems II. The lab introduces the C/S (Cybership) Enterprise, which is a model scale tug boat fitted with a Raspberry Pi embedded computer and a ROS based control system.

1.1 Robot Operating Software

This section will provide a brief introduction to the main concepts and advantages of the operating software used in the Marine Cybernetics lab; the *Robot Operating System* (ROS).

Introduced in 2007, the Robot Operating System 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 of 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 illustrates the basic peer-to-peer communication of a ROS-system.

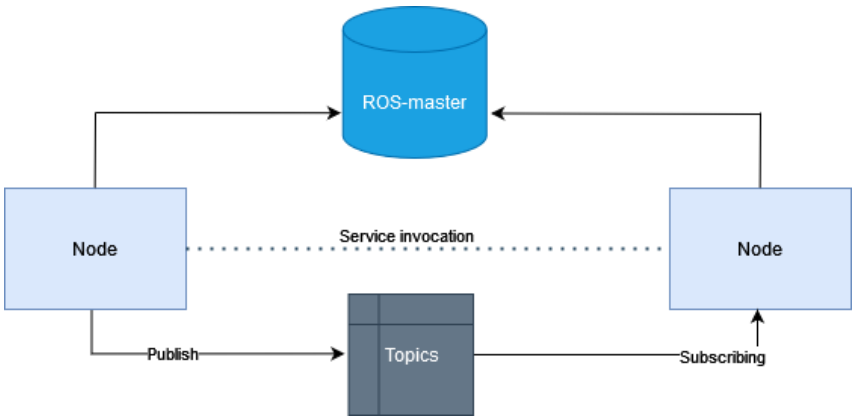

Figure 1: Basic ROS concept

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 you 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. As an open source project ROS offers a plethora of easy and accessible software for robots, making the integration of sensors a simple task. Most hardware comes with ROS-support either 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 mostly Python-based system.

2 Hardware

As part of the lab you will be provided with the following hardware:

1. Raspberry Pi 4b computer
2. A laptop running Ubuntu
3. Dualshock 4 wireless controller

3 Software Prerequisites

- Python 3.8.3+
- Visual Studio Code, with the remote SSH-extension
- A PC with an Ethernet-port or an ethernet-adapter.

The following are not required, but can be beneficial in work:

- A PC running Ubuntu
- PlotJuggler, a visualization tool
- bagpy, a tool for reading and plotting from ROS bag files.

The necessary equipment provided to you will come pre-installed with all the relevant software.

4 Networking

Connect the Raspberry Pi to your computer using an ethernet cable. Then, set the ethernet network settings on **your** computer to

- IP: 192.168.1.1
- Subnet mask: 255.255.255.0

A quick how-to for Windows 10 is found here. To verify the connection to raspberry pi, open up your preferred command-line tool and run the following command:

```
$ ping 192.168.1.2
```

where 192.168.1.2 is the IP of the Raspberry Pi. If the RPi responds you can SSH onto it and run files via the terminal. Simply run the command

```
$ ssh pi@192.168.1.2
```

Most likely you will then be prompted for a password. All the RPi's have ***marin33*** as the designated password.

5 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.

5.1 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 if you are using the RPi or the laptop. 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.

5.2 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.

5.3 The src-directory

This is the folder you will be working the most in, and were you will be putting all your ROS-packages. Templates for the ROS-packages can be found on Github. You can copy indivudal packages from this repository, or download the whole workspace as one. A typical workspace for the DP-lab should look something like this:

```
my_folder
├── build
├── devel
└── src
    ├── simulator
    │   ├── launch
    │   └── src
    │       └── CSEI.py
    ├── controller
    │   ├── launch
    │   └── src
    │       └── controller.py
    ├── messages
    └── common_tools
        ├── include
        └── setup.py
```

5.4 Running ROS nodes

After your code is written it is time two activate your ROS-nodes. We will cover how to write the code in a later section. This is simply to show how to run it.

5.4.1 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:

```
$ roscore
```

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

5.4.2 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 where you wrote your code, e.g "feedback_ctrl.py".

5.5 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 5.3.

Listing 1 shows the basic setup of a launch file. This example launches three nodes, the simulator, an observer and the controller-node.

To run a launch file we use the following command:

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

```
<launch>
    <node name="simulator"  pkg = "simulator" type="CSEI.py" />
    <node name="observer"   pkg = "observer"  type="observer.py" />
    <node name="controller" pkg="controller"  type="feedback_ctrl.py" />
</launch>
```

Listing 1: Example of a launch-file

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) .

5.6 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 pubishes/subscribed to in a ROS-system by using the command:

```
$ 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 would be called something like **CSEI/u**. If we wish to display the signals from this topic in real time we can run the command

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

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

5.7 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

```
$ 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

```
$ rosbag record CSEI/u CSEI/tau_cmd
```

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

5.8 Other usefull ROS-commands

- `$ rosnode list`

  - Shows a list of all active ROS-nodes

6 ROS templates

So that you do not have to spend to much time familiarizing yourself ROS, you will be provided with node templates and tools for publishing and subscribing to the relevant topics of each case study. These can be found in the Github repository of IMT.

6.1 Node-structure

Each ROS-package constitutes a component in a typical DP-system. Within every package there is a *src*-directory where a single python script is placed. This is where you will write and call your code. Listing 2 shows an example of a ROS-node that maps the inputs from a PS4 controller to forces in each of CSE1 thrusters.

Quickly summarized, you write your functions above the

```
if__name__ == 'main'
```

statment. These functions should then be called in the while loop located inside. The two statements before the while loop are functions for initiating the node, and setting the refresh rate, or frequency, that we want to run the node at. This is predefined in all templates so that you only have to worry about writing the functions and calling them inside the while loop.

NB! If you are experienced in ROS, or are motivated to learn, feel free to deviate from this structure and solve the Case Studies in a manner you see fit. The templates are there to save you time in the implementation. The only requirement is that you use Python as a programming language for your ROS-system. The example in this guide uses functional programming, but you can just as easily solve the problem object oriented. Using classes in ROS implementation is in fact often considered best practice.

6.2 Common_tools package

As seen in Listing 2, the templates import from a module called **common_tools**. This is where you can find all the tools for publishing and subscribing to the relevant

signals in your system, as well as the functions that initialize nodes. The tools are object-oriented, with every relevant signal pertaining to an object in the library. Each object contains methods called publish, and callback.

Every time a subscriber detects that a new signal is published to its topic, it calls the callback function in our object, and updates its class variables. You can then retrieve these by using a get-method, or just calling the variable directly through the object.

```
u = u_data.get_data() # Retrieve actuator commands
u = u_data.Udata      # Retrieve acutator commands
```

The publish method takes an input, and publishes it to the relevant topic.

```
u_data.publish(u) # Publish actuator commands
```

All objects are declared and imported into the relevant code-templates from the **lib** submodule . Subscriptions to the different topics are all handled inside these classes and the initialization nodes.

It is important that you at any time keep the common_tools package inside your workspace. The package also contains a submodule called **math_tools**. This contains some predefined mathematical functions that may be useful for the project. Rotation matrices and conversions between euler-angles and quaternions are examples of tool you can find here. Feel free to add more functions here as you please, and import them to your nodes. This makes for cleaner and more structured code.

7 PlotJuggler

To visualize the vessel position during simulation, we will be using a tool called PlotJuggler. This is program with with ROS-compatibility which allows you to view and visualize data in ROS-topics live.

To open the ROS-version of PlotJuggler run the command

```
$ rosrun plotjuggler plotjuggler
```

This will open a window that looks something like this:

In the dropdown menu under *Streaming*, pick **ROS topic subscriber**. Make sure you are connected to the RPi via ethernet, and click start. If you are prompted for the location of the ROS_MASTER, set it to: *http://192.168.1.2:11311*. You may have to do this in the command window before you start Plotjuggler as well. Then the command is:

```
$ export ROS_MASTER_URI=http://192.168.1.2:11311
```

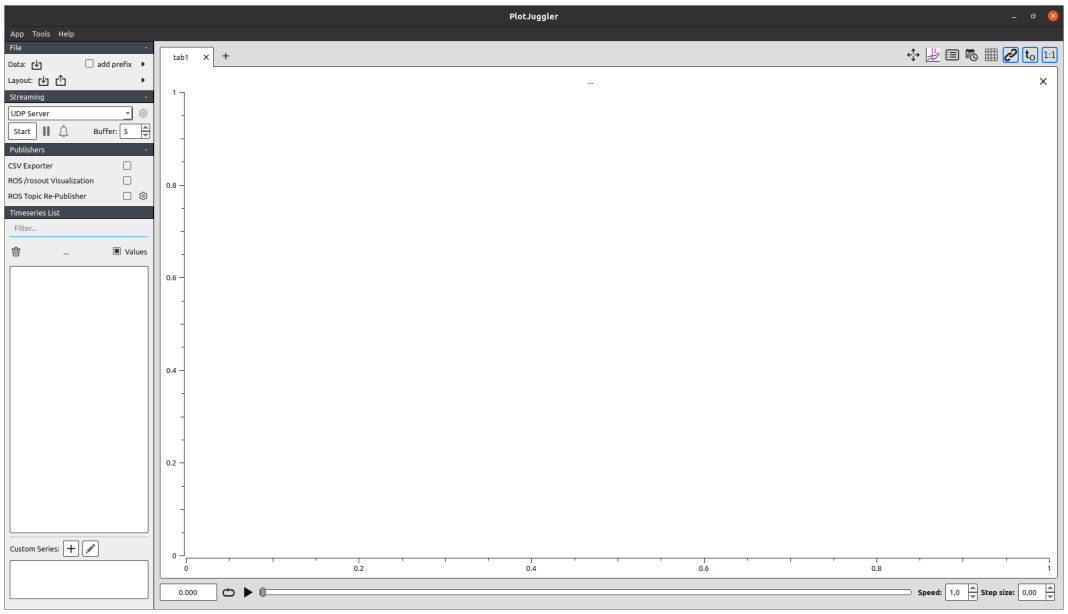

Figure 2: Plotjuggler

After pressing start you will get a list of available topics. Pick the relevant ones and pull them into the coordinate system to view the live feedback.

8 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 simulation every time you want to change them. To activate the node, and GUI use the following steps:

- Make sure you have imported the Gains object from lib.py. It is built like most other objects and contais get-methods for extracting gains, along with callback-functions.
- In the **observerInitNode()** or **controllerInitNode** uncomment the line starting with *gain_client*.
- To 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 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

- In another terminal, activate the GUI:

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

This should open a program that looks like this:

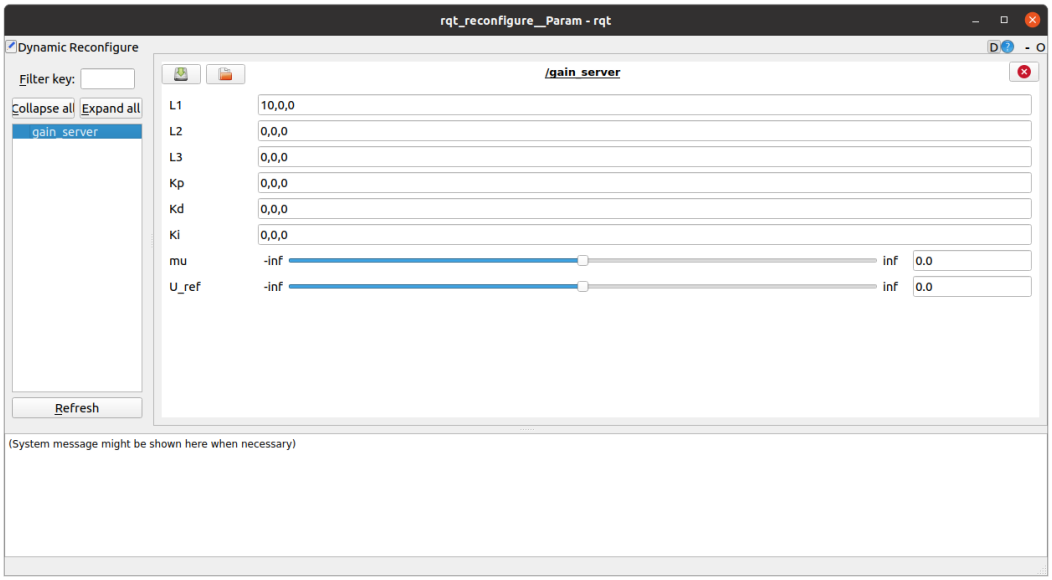

Figure 3: Tuning GUI

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.

```python
#!/usr/bin/env python3
import rospy
import numpy as np
import math
from nav_msgs.msg import Odometry
from std_msgs.msg import Float64MultiArray
from common_tools.lib import ps4, tau, u_data, controllNodeInit, nodeEnd
from common_tools.math_tools import *

### Write your code here ###

# Example functions for manuevering with the DS4 controller
def saturate(u):
    """
    Saturate ensures that the input to the actuator remains bounded to the interval [-1, 1
    """
    if u > 1:
        u = 1
    elif u < -1:
        u = -1
    return u

def sixaxis2thruster(lStickX, lStickY, rStickX, rStickY, R2, L2):
    """
    sixaxis2thruster() directly maps the sixaxis playstation controller inputs
    to the vessel actuators.
    """
    ### Acutator commands ###
    u1 = -0.5*(L2 - R2)
    u2 = saturate(math.sqrt(lStickX ** 2  + lStickY ** 2))
    u3 = saturate(math.sqrt(rStickX ** 2 + rStickY ** 2))


    ### VSD angles as described in the handbook ###
    alpha1 = math.atan2(lStickX, lStickY)
    alpha2 = math.atan2(rStickX, rStickY)

    u = np.array([u1, u2, u3, alpha1, alpha2])
    return u



if __name__ == '__main__':

    node = controllNodeInit()
    r = rospy.Rate(100)

    while not rospy.is_shutdown():
            # Handle calls to methods or functions as below
        u = sixaxis2thruster(ps4.lStickX, ps4.lStickY, ps4.rStickX, ps4.rStickY, ps4.R2,
        u_data.publish(u)
        r.sleep()

    nodeEnd(node)
```


Listing 2: Example of a node-file