RISC-Docs Documentation

Release 0.1

RISC Members

Mar 23, 2022

Aerial Bootcamp

Note: This guide is under active development.

This document provides tutorials/guides/manuals of setups and experiments in the RISC Lab. This is mainly intended for the Robotics, Intelligent Systems and Control Lab users, at KAUST.

CHAPTER 1

ROS Basics

1.1 Main Objectives of this Tutorial

1. The objective of this course is to give you the basic tools and knowledge to be able to understand and create any basic ROS related project. You will be able to **move robots, read their sensor data, make the robots perform intelligent tasks, see visual representations of complex data such as laser scans and debug errors in the programs.**
2. The course will allow you to **understand packages that others have done**. So you can take ROS code made by others and understand what is happening and how to modify it for your own purposes
3. This course can serve as an introduction to be able to understand the ROS documentation of complex ROS packages for object recognition, text to speech, navigation and all the other areas where has ROS developed code.

What is presented in this document is the main ROS concepts that are the core of ROS. These are the most important concepts that you have to master. Once you master them, the rest of ROS can follow easily.

Along the parts of this course, you will learn:

- How **ROS Basic Structure** works.

- What are **ROS Topics** and how to use them.
- What are **ROS Services** and how to use them.
- What are **ROS Actions** and how to use them.
- How to use **ROS Debugging Tools** for finding errors in your programs (especially Rviz).

Note: We will use **Python** as the programming language in all the course exercises

---

Important: **DO NOT SKIP EXERCISES**. Exercises are the core of this tutorial (remember, practice, practice, practice). If you avoid them, you will be missing the whole thing.

1.2 Basic Concepts

1.2.1 What is ROS?

ROS is a software framework for writing robot software. The main aim of ROS is to reuse the robotic software across the globe. ROS consists of a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.

Official definition from ROS Wiki:

> ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

In this tutorial, we are going to work with a specific version of ROS called Melodic. Also, some ROS packages are needed in order to perform the simulation exercises mentioned in this tutorial. The following sections will guide you through the installation procedures.

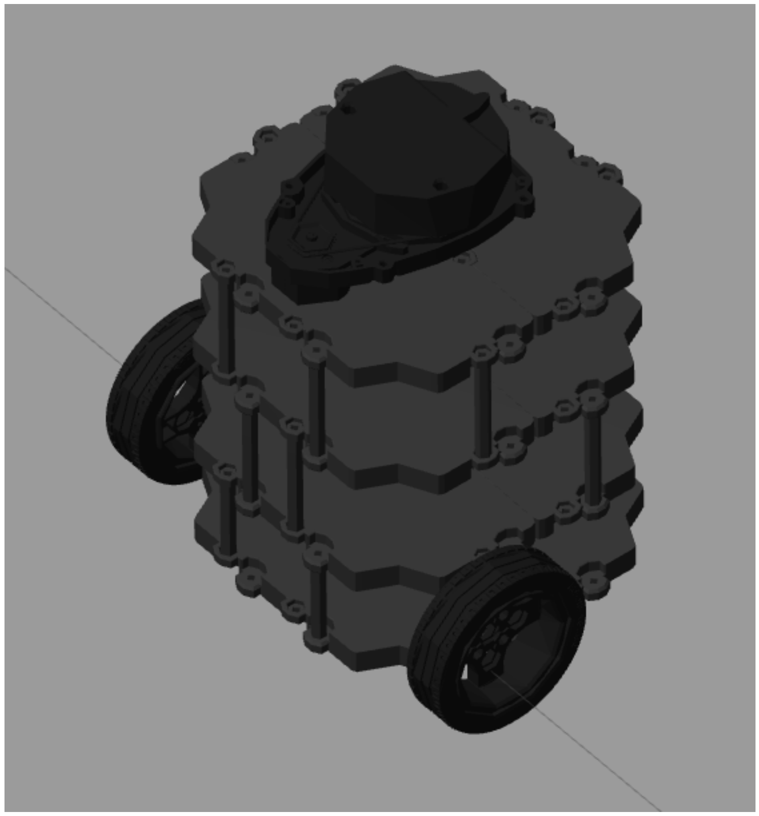
1.2.2 System Setup

Assuming you have a clean install of Ubuntu 20.04, follow the steps from the following video.

https://youtu.be/8AeTQ8Ew0hc

Install TurtleBot packages

During this tutorial, you will work with a simulated robot called **TurtleBot3**, to apply the concepts of ROS. The following image is a picture of the robot you will work with. It is a differential drive robot, that has a Kinect sensor for environmental mapping, wheel encoders for position estimation.



For reference see Turtlebot wiki page.

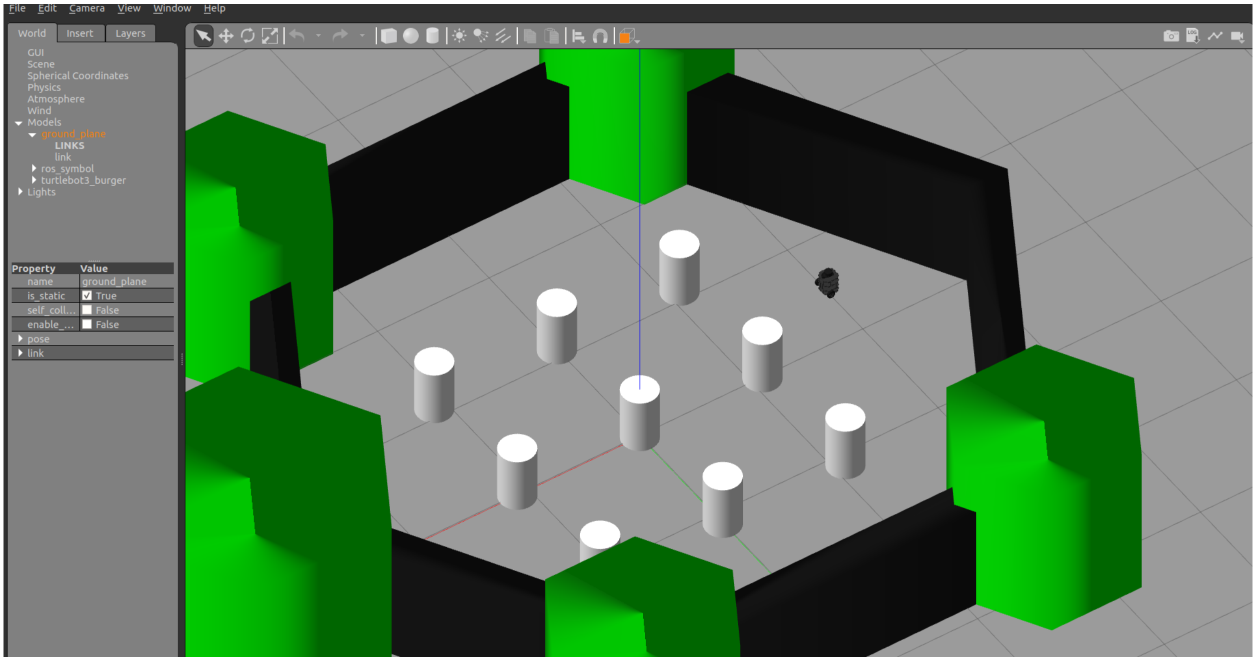
Open application called **Terminator** (you can install it by running following command in the terminal `sudo apt-get install terminator`), it's highly recommended to use this application instead of stock Terminal. You can have tabs or split windows into few terminals. To install the required packages, execute the following commands.

```
sudo apt-get install ros-melodic-turtlebot3* -y
echo "export TURTLEBOT3_MODEL=burger" >> ~/.bashrc
```

After installation is done, check that the simulation works in Gazebo. Execute the following commands in a shell terminal.

```
roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

You should get something similar to the following.



1.2.3 Move the robot

How can you move the Turtlebot?

The easiest way is by executing an existing ROS program to control the robot. A ROS program is executed by using some special files called **launch files**. Since a previously-made ROS program already exists that allows you to move the robot using the keyboard, let's launch that ROS program to teleoperate the robot.

Execute in a separate terminal:

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

Read the instructions on the screen to know which keys to use to move the robot around, and start moving the robot!

Try it!! When you're done, you can `Ctrl+C` to stop the execution of the program.

So, you used a command called `roslaunch`. What is that command?

`roslaunch` is the command used to launch a ROS program. It's structure goes as follows:

```
roslaunch <package_name> <launch_file>
```

As you can see, that command has two parameters: the first one is **the name of the package** that contains the launch file, and the second one is **the name of the launch file** itself (which is stored inside the package).

Now, what is a package?!

1.2.4 What is a package?

ROS uses **packages** to organize its programs. You can think of a package as **all the files that a specific ROS program contains**; all its cpp files, python files, configuration files, compilation files, launch files, and parameters files. All those files in the package are organized with the following structure:

- **launch** folder: Contains launch files

- **src** folder: Source files (cpp, python)
- **CMakeLists.txt**: List of cmake rules for compilation
- **package.xml**: Package information and dependencies

To go to any ROS package, ROS gives you a command named `roscd`. When typing:

```
roscd <package_name>
```

It will take you to the path where the package *package_name* is located.

Example: navigate to the `turtlebot_teleop` package, and check that it has that structure.

```
roscd turtlebot_teleop
ls
```

`roscd` is a command which will get you to a ROS package location. `ls` is a command that lists the content of a folder.

- Every ROS program that you want to execute is organized in a package.
- Every ROS program that you create will have to be organized in a package.
- Packages are the main organization system of ROS programs.

1.2.5 What is a launch file ?

We've seen that ROS uses launch files in order to execute programs. But… how do they work? Let's have a look.

lets have a look at a launch file. Open the launch folder inside the `turtlebot_teleop` package and check the `keyboard_teleop.launch` file.

```
roscd turtlebot_teleop
cd launch
cat keyboard_teleop.launch
```

You will see:

```
<launch>
  <!-- turtlebot_teleop_key already has its own built in velocity smoother -->
  <node pkg="turtlebot_teleop" type="turtlebot_teleop_key" name="turtlebot_teleop_
↪keyboard"  output="screen">
    <param name="scale_linear" value="0.5" type="double"/>
    <param name="scale_angular" value="1.5" type="double"/>
    <remap from="turtlebot_teleop_keyboard/cmd_vel" to="cmd_vel_mux/input/teleop"/>
  </node>
</launch>
```

In the launch file, you have some extra tags for setting parameters and remaps. For now, don't worry about those tags and focus on the node tag.

All launch files are contained within a `<launch>` tag. Inside that tag, you can see a `<node>` tag, where we specify the following parameters:

- pkg="`package_name`": Name of the package that contains the code of the ROS program to execute
- type="`python_file_name.py`" : Name of the program file that we want to execute
- name="`node_name`" : Name of the ROS node that will launch our Python file
- output="`type_of_output`" : Through which channel you will print the output of the Python file

1.2.6 Create a package

Until now we've been checking the structure of an already-built package. But now, lets create one ourselves. When we want to create packages, we need to work in a very specific ROS workspace, which is known as the **catkin workspace**. The **catkin workspace** is the directory in your hard disk where your own ROS packages must reside in order to be usable by ROS. Usually, the catkin workspace directory is called *catkin_ws*.

- catkin_ws

Usually, the *catkin_ws* is created in the *home* folder of your user account. We've already created and initialized catkin workspace for you.

Go to the *src* folder inside *catkin_ws*

```
cd ~/catkin_ws/src
```

The *src* directory is the folder which holds created packages. Those could be your own packages, or packages that you copied from other sources e.g. *Github* repository.

In order for the ROS system to recognize the packages in your *catkin_ws*, it needs to be on the ROS file path. ROS file path is an Ubuntu environment variable that holds the paths to ROS packages. To add our catkin_ws to the ROS file path follow the following instructions.

First, build (compile) your workspace. It's OK to build the catkin_ws even if it has no packages. After the build process, some new folders will appear inside your catkin_ws. One of the folders, called *catkin_ws/devel* contains a setup file which will be used to add the path of the catkin_ws to the ROS file path. Build the *catkin_ws* using the `catkin build` inside the *catkin_ws*:

```
# navigate to the catkin_ws
cd ~/catkin_ws
# build
catkin build
```

Now, let's add the *catkin_ws* path. Execute the following command while being inside *catkin_ws*

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

This will add the *catkin_ws* path in the current terminal session. Once you close the terminal window, it forgets it! So, you will have to do it again each time you open a terminal in order for ROS to recognize your workspace! Yah, I know, that sucks! But no worries, there is a solution. You can automate the execution of the above command each time you open a terminal window. To do that, you want to add the above command to a special file called `.bashrc` that is located inside your home folder.

```
# go to the home folder
cd ~
# open the .bashrc file
nano .bashrc
```

add the command `source ~/catkin_ws/devel/setup.bash` to the end of `.bashrc`. Then, hit `CTRL+x`, then, `y`, to save the changes to the file.

Now, lets create a package.

Important: Remember to create ROS packages inside the `src` folder

Create a package

```
catkin_create_pkg my_package rospy
```

This will create, inside our `src`, directory a new package with some files in it. We'll check this later. Now, let's see how this command is built:

```
catkin_create_pkg <package_name> <package_dependecies>
```

The **package_name** is the name of the package you want to create, and the **package_dependencies** are the names of other ROS packages that your package depends on.

Now, re-build your catkin_ws and source it as above.

In order to check that our package has been created successfully, we can use some ROS commands related to packages. For example, let's type:

```
rospack list
rospack list | grep my_package
roscd my_package
```

`rospack list`: Gives you a list with all of the packages in your ROS system. `rospack list | grep my_package`: Filters, from all of the packages located in the ROS system, the package named *my_package*. `roscd my_package`: Takes you to the location in the Hard Drive of the package, named *my_package*.

1.2.7 Your First ROS Program

At this point, you should have your first package created... but now you need to do something with it! Let's do our first ROS program!

1. Create in the `src` directory in `my_package` a python file that will be executed. For this exercise, just copy this simple python code `simple.py` below.

```
#! /usr/bin/env python
# The previous line will ensure the interpreter used is the first one on your␣
↪environment's $PATH. Every Python file needs to start with this line at the top.

import rospy # Import the rospy, which is a Python library for ROS.

rospy.init_node('simple_node') # Initiate a node called ObiWan

print "Help me buddy, you are my only hope" # A simple Python print
```

2. Save the file. You will need to make this file executable by using the `chmod` linux command as follows.

```
# navigate to the src folder inside my_package
roscd my_package/src
# make the python file executable
chmod +x simple.py
```

3. Create a `launch` directory inside the package named `my_package`

```
roscd my_package
# the following command creates a directory
mkdir launch
```

4. Create a new launch file inside that launch directory

```
gedit launch/my_package_launch_file.launch
```

5. Fill this launch file as we've previously seen in the launch file of the `turtlebot_teleop` package,

```
<launch>
  <!-- turtlebot_teleop_key already has its own built in velocity smoother -->
  <node pkg="turtlebot_teleop" type="turtlebot_teleop_key" name="turtlebot_teleop_
↪keyboard"  output="screen">
    <param name="scale_linear" value="0.5" type="double"/>
    <param name="scale_angular" value="1.5" type="double"/>
    <remap from="turtlebot_teleop_keyboard/cmd_vel" to="cmd_vel_mux/input/teleop"/>
  </node>
</launch>
```

6. Modify the launch file to run your ROS program `simple.py`

```
<launch>
 <!-- run simple.py from my_package -->
 <node pkg="my_package" type="simple.py" name="simple_node"  output="screen">
 </node>
</launch>
```

7. Finally, execute the `roslaunch` command in the terminal in order to launch your program.

```
roslaunch my_package my_package_launch_file.launch
```

You should see the print statement

```
Help me body, you are my only hope
```

Hint: Usually, when we add ROS program to a package, we re-build the *catkin_ws* and source it. However, since we are working with `Python`, we will not need to do that for now, because a Python code does not need to compile. If you write a `C++` ROS program, then, you will need to re-build your *catkin_ws*.

1.2.8 ROS Nodes

You've initiated a node in the previous code but… what's a node? ROS nodes are basically programs made in ROS. The ROS command to see what nodes are actually running in a computer is:

```
rosnode list
```

Type the previous command in a new terminal and look for the node you've just initiated `simple_node`.

You can't find it? I know you can't. That's because the node is killed when the Python program ends. Let's change that.

Update your Python file `simple.py` with the following code:

```
#! /usr/bin/env python

import rospy

rospy.init_node("simple_node")
rate = rospy.Rate(2)               # We create a Rate object of 2Hz
```

(continues on next page)

(continued from previous page)

```
while not rospy.is_shutdown():      # Endless loop until Ctrl + C
   print "Help me body, you are my only hope"
   rate.sleep()                     # We sleep the needed time to maintain the Rate␣
↪fixed above

# This program creates an endless loop that repeats itself 2 times per second (2Hz)␣
↪until somebody presses Ctrl + C in the Shell
```

Launch your program again using the `roslaunch` command.

```
roslaunch my_package my_package_launch_file.launch
```

Now try again in another terminal window:

```
rosnode list
```

Can you now see your node? You should be!

In order to see information about our node, we can use the next command:

```
rosnode info /simple_node
```

This command will show us information about all the connections that our Node has.

1.2.9 Parameters Server

A Parameter Server is a dictionary that ROS uses to store parameters. These parameters can be used by nodes at runtime and are normally used for static data, such as configuration parameters.

To get a list of these parameters, you can type:

```
rosparam list
```

To get a value of a particular parameter, you can type:

```
rosparam get <parameter_name>
```

And to set a value to a parameter, you can type:

```
rosparam set <parameter_name> <value>
```

You can create and delete new parameters for your own use, but do not worry about this right now. You will learn more about this later.

1.2.10 ROSCORE

In order to have all of this working, we need to have a roscore running. The roscore is the main process that manages all of the ROS system. You always need to have a roscore running in order to work with ROS. The command that launches a roscore is:

```
roscore
```

So, this is the first command that should be executed before using other ROS functionalities.

Hint: When you use `roslaunch` to run your ROS nodes, it automatically runs `roscore` if it is not already run.

1.2.11 Environment Variables

ROS uses a set of Linux system environment variables in order to work properly. You can check these variables by typing:

```
export | grep ROS
```

You will get something similar to:

```
user ~ $ export | grep ROS
declare -x ROSLISP_PACKAGE_DIRECTORIES="/home/user/catkin_ws/devel/share/common-lisp"
declare -x ROS_DISTRO="melodic"
declare -x ROS_ETC_DIR="/opt/ros/melodic/etc/ros"
declare -x ROS_MASTER_URI="http://localhost:11311"
declare -x ROS_PACKAGE_PATH="/home/user/catkin_ws/src:/opt/ros/melodic/share:/opt/ros/
↪melodic/stacks"
declare -x ROS_ROOT="/opt/ros/melodic/share/ros"
```

The most important variables are the **ROS_MASTER_URI** and the **ROS_PACKAGE_PATH**.

ROS_MASTER_URI: Contains the url where the ROS Core is being executed. Usually, your own computer (localhost). **ROS_PACKAGE_PATH**: Contains the paths in your Hard Drive where ROS has packages in it.

1.2.12 Summary

So, now, what is ROS again?

ROS is basically the framework that allows us to do all that we showed along this chapter. It provides the background to manage all these processes and communications between them… and much, much more!! In this tutorial you've just scratched the surface of ROS, the basic concepts. ROS is an extremely powerful tool. If you dive into our courses you'll learn much more about ROS and you'll find yourself able to do almost anything with your robots!

Next we will start to talk about ROS topics, services, actions,and finally some debugging tools.

1.3 ROS Topics

What will you learn with this part?

- What are ROS topics and how to manage them?
- What is subscribers and publisher and how to create them?
- What are topic messages and how they work?

We will start by learning about a publisher.

1.3.1 What is a Publisher?

Let's create a ROS node that uses a publisher to publish some data. In the `src` folder of your package `my_package`, create the following node, and name it `simple_node_publisher.py`:

```
#! /usr/bin/env python

import rospy                                    # Import the Python library for ROS
from std_msgs.msg import Int32                  # Import the Int32 message from the std_
↪msgs package

rospy.init_node('topic_publisher')              # Initiate a Node named 'topic_publisher'
pub = rospy.Publisher('counter', Int32)         # Create a Publisher object, that will␣
↪publish on the /counter topic
                                                #  messages of type Int32

rate = rospy.Rate(2)                            # Set a publish rate of 2 Hz
count = Int32()                                 # Create a var of type Int32
count.data = 0                                  # Initialize 'count' variable

while not rospy.is_shutdown():                  # Create a loop that will go until someone␣
↪stops the program execution
  pub.publish(count)                            # Publish the message within the 'count'␣
↪variable
  count.data += 1                               # Increment 'count' variable
  rate.sleep()                                  # Make sure the publish rate maintains at␣
↪2 Hz
```

Use what you know about launch files to create a launch file to run this node. Let the launch file name be `launch_publisher.launch`. Run the launch file using `roslaunch`

You have just created a topic named `/counter`, and published through it as an integer that increases indefinitely. Wait! What is a topic?.

ROS Topic: A topic is like a pipe. **Nodes use topics to publish information for other nodes** so that they can communicate. You can find out, at any time, the number of topics in the system by doing a `rostopic list`. You can also check for a specific topic.

Now, given that you are still running the node you just created, execute the following command in a new terminal window.

```
rostopic list | grep  '/counter'
```

Here, you have just listed all of the topics running right now and filtered with the **grep** command the ones that contain the word `/counter`. If it appears, then the topic is running as it should.

You can request information about a topic by doing `rostopic info <name_of_topic>'`.

Now, type

```
rostopic info /counter
```

You should get something like this

```
Type: std_msgs/Int32
Publishers:
 * /topic_publisher (http://ip-172-31-16-133:47971/)
Subscribers: None
```

The output indicates the type of information published `std_msgs/Int32`, the node that is publishing `/topic_publisher`, and if there is a node listening to that info (None in this case).

Now, let's check the output of the `/counter` topic

```
rostopic echo /counter
```

You should see a succession of consecutive numbers, similar to the following

```
rostopic echo /counter
data:
985
---
data:
986
---
data:
987
---
data:
988
---
```

So, what has just happened? Go back and take a look at the comments in the last code.

So basically, what this code does is to **initiate a node and create a publisher that keeps publishing into the ''/counter'' topic a sequence of consecutive integers**.

Summarizing:

- **A publisher is a node that keeps publishing a message into a topic**. So now. . . what's a topic?
- **A topic is a channel that acts as a pipe, where other ROS nodes can either publish or read information**. Let's now see some commands related to topics (some of them you've already used).
- To **get a list of available topics** in a ROS system, you have to use the next command:

```
rostopic list
```

To **read the information that is being published in a topic,** use the next command:

```
rostopic echo <topic_name>
```

This command will start printing all of the information that is being published into the topic, which sometimes (ie: when there's a massive amount of information, or when messages have a very large structure) can be annoying. In this case, you can **read just the last message published into a topic** with the next command:

```
rostopic echo <topic_name> -n1
```

To **get information about a certain topic,** use the next command:

```
rostopic info <topic_name>
```

Finally, you can check the different options that `rostopic` command has by using the next command:

```
rostopic -h
```

1.3.2 ROS Messages

As you may have noticed, topics handle information through messages. There are many different types of messages.

In the case of the code you executed before, the message type was an `std_msgs/Int32`, but ROS provides a lot of different messages. You can even create your own messages, but it is recommended to use ROS default messages when its possible.

Messages are defined in **<name>.msg** files, which are located inside a `msg` directory of a package.

To **get information about a message,** you use the next command:

```
rosmsg show <message>
```

For example, let's try to get information about the `std_msgs/Int32` message. Type the following command and check the output.

```
rosmsg show std_msgs/Int32
```

You should get something like

```
[std_msgs/Int32]:
int32 data
```

In this case, the `Int32` message has only one variable named `data` of type `int32`. This `Int32` message comes from the package `std_msgs`, and you can find it in its `msg` directory. If you want, you can have a look at the `Int32.msg` file by executing the following command:

```
roscd std_msgs/msg
```

1.3.3 Exercise: Move the Robot

Now you're ready to create your own publisher and make the robot move, so let's go for it!

Create a launch file that launches the code `simple_topic_publisher.py` (you should have already done that in a previous step)

Modify the code you used previously to publish data to the `cmd_vel_mux/input/teleop` topic.

Launch the program and check that the robot moves.

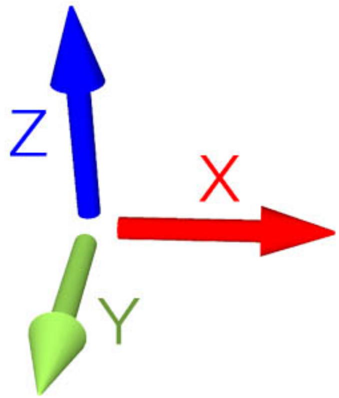
Hint: First, you need to bring up the robot simulation in Gazebo.

The `cmd_vel_mux/input/teleop` topic is the topic used to move the robot. Do a `rostopic info cmd_vel_mux/input/teleop` in order to get information about this topic, and identify the message it uses. You have to modify the code to use that message.

In order to fill the Twist message, you need to create an instance of the message. In Python, this is done like this: `var = Twist()`

In order to know the structure of the Twist messages, you need to use the `rosmsg show` command, with the type of the message used by the topic `cmd_vel_mux/input/teleop`.

In this case, the robot uses a differential drive plugin to move. That is, the robot can only move linearly in the x axis, or rotationally in the angular z axis. This means that the only values that you need to fill in the Twist message are the linear x and the angular z.



The magnitudes of the Twist message are in m/s, so it is recommended to use values between 0 and 1. For example, *0.5 m/s*

Solution to the exercise is available, **but** try to do it yourself and fight for it!

1.3.4 ROS Subscriber

You've learned that a topic is a channel where nodes can either write or read information. You've also seen that you can write into a topic using a publisher, so you may be thinking that there should also be some kind of similar tool to read information from a topic. And you're right! That's called a subscriber. **A subscriber is a node that reads information from a topic**. Let's create a subscriber node.

Important: Make sure that you terminated all terminal sessions before you continue

---

Create a Python node named `simple_topic_subscriber.py` and copy the following code

```
#! /usr/bin/env python

import rospy
from std_msgs.msg import Int32

def callback(msg):                                    # Define a function called
↪'callback' that receives a parameter
                                                      # named 'msg'

  print msg.data                                      # Print the value 'data' inside␣
↪the 'msg' parameter

rospy.init_node('topic_subscriber')                   # Initiate a Node called 'topic_
↪subscriber'

sub = rospy.Subscriber('/counter', Int32, callback)   # Create a Subscriber object␣
↪that will listen to the /counter
                                                      # topic and will cal the
↪'callback' function each time it reads
                                                      # something from the topic
rospy.spin()                                          # Create a loop that will keep␣
↪the program in execution
```

Save the node.

Important: Don't forget to give execution permission to the node using `chmod` command

As you did for the publisher node, create a *launch* file named `subscriber_launch.launch`, in the launch folder, which launches this node.

Run the launch file using `roslaunch my_package subscriber_launch.launch` command

What's up? Nothing happened again? Well, that's not actually true... Let's do some checks.

- Go to a new terminal and execute

```
rostopic echo /counter
```

You should see an output like

```
WARNING: no messages received and simulated time is active.
Is /clock being published?
```

And what does this mean? This means that nobody is publishing into the `/counter` topic, so there's no information to be read. Let's then publish something into the topic and see what happens.

For that, let's introduce a new command:

```
rostopic pub <topic_name> <message_type> <value>
```

This command will publish the message you specify with the value you specify, in the topic you specify.

Open a new terminal window (leave the one with the `rostopic echo` opened) and type the next command

```
rostopic pub /counter std_msgs/Int32 5
```

You will see something similar to the following

```
WARNING: no messages received and simulated time is active.
Is /clock being published?
data:
5
---
```

This means that the value you published has been received by your subscriber program (which prints the value on the screen).

So now, let's explain what has just happened. You've basically created a subscriber node that listens to the `/counter` topic, and each time it reads something, it calls a function that does a print of the msg. Initially, nothing happened since nobody was publishing into the `/counter` topic, but when you executed the `rostopic pub` command, you published a message into the `/counter` topic, so the function has printed the number and you could see that message in the `rostopic echo` output. Now everything makes sense, right? I hope!

Now let's do some exercises to put into practice what you've learned!

1.3.5 Exercise: Print Robot's Odometry

Modify the code in the publisher node in order to print the odometry of the robot.

The odometry of the robot is published by the robot into the `/odom` topic.

You will need to figure out what message uses the `/odom` topic, and how the structure of this message is.

Solution is available, but try yourself and fight for it!

1.3.6 Exercise: Publishing to Custom Message

Create a python file (e.g. `publish_age.py`) that creates a publisher which publishes the age of the robot, to the previous package.

For that, you'll need to create a new message called `Age.msg`. See the detailed description below on how to prepare `CMakeLists.txt` and `package.xml` for custom topic message compilation.

Solution is available,**but** try yourself and fight for it!

1.3.7 Creating Custom Messages

Now you may be wondering… in case I need to publish some data that is not an `Int32`, which type of message should I use? You can use all ROS defined (`rosmsg list`) messages. But, in case none fit your needs, you can create a new one.

In order to create a new message, you will need to do the following steps:

Create a directory named `msg` inside your package, e.g. `my_package/msg`

Inside this directory, create a file named `Name_of_your_message.msg` (more information down)

Modify `CMakeLists.txt` file (more information down)

Modify `package.xml` file (more information down)

Compile

Use in code

For example, let's create a message that indicates age, with years, months, and days.

Create a directory msg in your package.

```
roscd my_package
mkdir msg
```

Add the `Age.msg` file which must contain this:

```
float32 years
float32 months
float32 days
```

Save it.

In `CMakeLists.txt` of your package, you will have to edit four functions.

- find_package()
- add_message_files()
- generate_messages()
- catkin_package()

`find_package()`

This is where all the packages required to COMPILE the messages of the topics, services, and actions go. In package.xml, you have to state them as `build_depend`.

Hint: If you open the `CMakeLists.txt` file in your IDE, you'll see that almost all of the file is commented. This includes some of the lines you will have to modify. Instead of copying and pasting the lines below, find the equivalents in the file and uncomment them, and then add the parts that are missing.

```
find_package(catkin REQUIRED COMPONENTS
       rospy
       std_msgs
       message_generation   # Add message_generation here, after the other packages
)
```

`catkin_package()`

State here all of the packages that will be needed by someone that executes something from your package. All of the packages stated here must be in the package.xml as `run_depend`.

```
catkin_package(
      CATKIN_DEPENDS rospy message_runtime   # This will NOT be the only thing here
)
```

`add_message_files()`

This function includes all of the messages of this package (in the `msg` folder) to be compiled. The file should look like this.

```
add_message_files(
      FILES
      Age.msg
    ) # Don't Forget to uncomment the parenthesis and add_message_files TOO
```

`generate_messages()`

Here is where the packages needed for the messages compilation are imported.

```
generate_messages(
      DEPENDENCIES
      std_msgs
) # Dont Forget to uncomment here too
```

In summary, this is the minimum expression of what is needed for the `CMakeLists.txt` to work:

```
cmake_minimum_required(VERSION 2.8.3)
project(my_package)



find_package(catkin REQUIRED COMPONENTS
  std_msgs
  message_generation
)

 add_message_files(
   FILES
   Age.msg
 )

 generate_messages(
```

(continues on next page)

(continued from previous page)

```
  DEPENDENCIES
  std_msgs
 )


catkin_package(
  CATKIN_DEPENDS rospy message_runtime
)

include_directories(
  ${catkin_INCLUDE_DIRS}
)
```
```

Modify `package.xml` by adding these 2 lines.

```xml
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

This is the minimum expression of the `package.xml`

```xml
<?xml version="1.0"?>
<package>
  <name>my_package</name>
  <version>0.0.0</version>
  <description>The my_package package</description>

  <maintainer email="user@todo.todo">user</maintainer>

  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>rospy</build_depend>
  <build_depend>message_generation</build_depend>
  <run_depend>rospy</run_depend>
  <run_depend>message_runtime</run_depend>

  <export>

  </export>
</package>
```

Now you have to compile the msgs. To do this, you have to type in the terminal,

```bash
cd ~/catkin_ws
catkin build
source devel/setup.bash
```

Warning: When you compile new messages, there is still an extra step before you can use the messages. You have to type in the terminal, in the *catkin_ws*: `source devel/setup.bash`. This executes this bash file that sets, among other things, the newly generated messages created through the `catkin build`. If you don't do this, it might give you a python import error, saying it doesn't find the message generated.

Hint: To verify that your message has been created successfully, type in your terminal `rosmsg show Age`. If the
```

structure of the Age message appears, it will mean that your message has been created successfully and it's ready to be used in your ROS programs.

Execute in a terminal

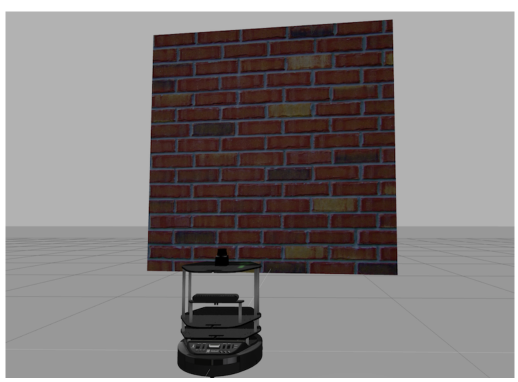
```
rosmsg show Age
```

You should get

```
[my_package/Age]:
float32 years
float32 months
float32 days
```

> **Warning:** There is an issue in ROS that could give you problems when importing msgs from the `msg` directory. **If your package has the same name as the Python file that does the import of the msg**, this will give an error saying that it doesn't find the msg element. This is due to the way Python works. Therefore, you have to be careful to **not name the Python file exactly the same as its parent package**.
>
> Example:
>
> Package name = `my_package`
>
> Python file name = `my_package.py`
>
> This will give an import error because it will try to import the message from the `my_package.py` file, from a directory `.msg` that doesn't exists.

1.3.8 Project



With all you've learned during this course, you're now able to do a small project to put everything together. Subscribers, Publisher, Messages… you will need to use all of this concepts in order to execute the following mini project!

In this project, you will create a code to make the robot avoid the wall that is in front of it. To help you achieve this, let's divide the project down into smaller units:

Create a Publisher that writes into the `cmd_vel_mux/input/teleop` topic in order to move the robot.

Create a Subscriber that reads from the `/scan` topic. This is the topic where the laser publishes its data.

Depending on the readings you receive from the laser's topic, you'll have to change the data you're sending to the `cmd_vel_mux/input/teleop` topic, in order to avoid the wall. This means, use the values of the laser to decide.

Hint: The data that is published into the `/scan` topic has a large structure. For this project, you just have to pay attention to the `ranges` array.

To check the laser message type, execute the following:

```
rosmsg show sensor_msgs/LaserScan
```

You should get

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges <-- Use only this one
float32[] intensities
```

Hint: The `ranges` array has a lot of values. The ones that are in the middle of the array represent the distances that the laser is detecting right in front of him. This means that the values in the middle of the array will be the ones that detect the wall. So in order to avoid the wall, you just have to read these values.

Hint: The laser has a range of 30m. When you get readings of values around 30, it means that the laser isn't detecting anything. If you get a value that is under 30, this will mean that the laser is detecting some kind of obstacle in that direction (the wall).

Hint: The scope of the laser is about 180 degrees from right to left. This means that the values at the beginning and at the end of the `ranges` array will be the ones related to the readings on the sides of the laser (left and right), while the values in the middle of the array will be the ones related to the front of the laser.

So, in the end, you probably will get something like the following:

The robot moves forward until it detects an obstacle in front of it which is closer than 1 meter, so it begins to turn left in order to avoid it.

The robot keeps turning left and moving forward until it detects that it has an obstacle at the right side which is closer than 1 meter, so it stops and turns left in order to avoid it.

1.4 ROS Services

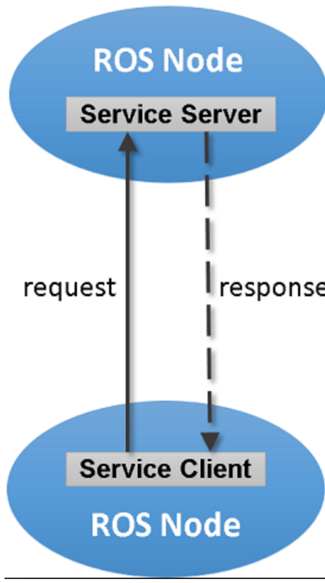
Services are another way that nodes can communicate with each other. Services allow nodes to send a **request** and receive a **response**.

As you have seen, ROS topics are means of communications between nodes, but they don't execute functionalities. They just hold data. Services, however, can provide a specific functionality once they receive a request to do so. For example, a service can provide the number of detected person in an image.

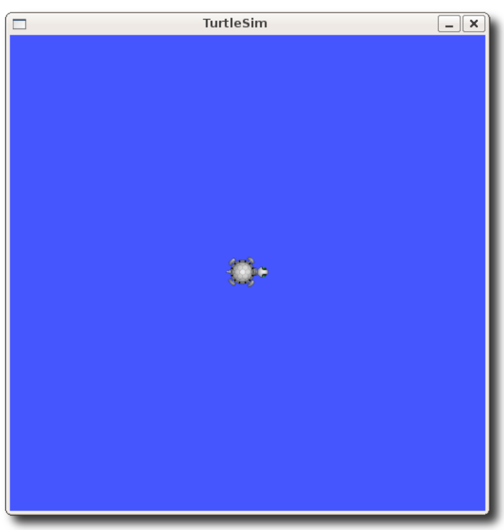
A service has two parts, **server** and **client**

A service **server** is a ROS program the implements certain functionality. Once it is executed, it will wait for a call from a **client**. Once a call use received, it will execute its functionality and provide a **response**.

A **client** uses some ROS commands to **request** a service from a service **server**



In this part, we are going to use a different simulation setup, a simpler one, called `turtlesim`



To install `turtlesim`

```
sudo apt-get install ros-melodic-turtlesim
```

To run the `turtlesim` node and control the turtle using keyboard, execute

```
# run roscore in a seperate terminal
roscore
# in a separate terminal, run the sim node
rosrun turtlesim turtlesim_node
# in a separate terminal, run the keyboard telep node
rosrun turtlesim turtle_teleop_key
```

The main ROS command used with services is called `rosservice`. The following some commands that can be used on service topics.

```
rosservice list         # print information about active services
rosservice call         # call the service with the provided args
rosservice type         # print service type
rosservice find         # find services by service type
rosservice uri          # print service ROSRPC uri
```

1.4.1 Command `rosservice list`

Now, lets check the available ROS services using the `rosservice` command

```
rosservice list
```

The `list` command shows us that the turtlesim node provides nine services: `reset`, `clear`, `spawn`, `kill`, `turtle1/set_pen`, `/turtle1/teleport_absolute`, `/turtle1/teleport_relative`, `turtlesim/get_loggers`, and `turtlesim/set_logger_level`. There are also two services related to the separate `rosout` node: `/rosout/get_loggers` and `/rosout/set_logger_level`. After executing the previous command, you will get some output like the following:

```
/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/spawn
/teleop_turtle/get_loggers
/teleop_turtle/set_logger_level
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level
```

Let's look more closely at the `clear` service using `rosservice type`:

1.4.2 Command `rosservice type`

The command can be used as follows:

```
rosservice type [service]
```

Let's try to find the type of `clear` service

```
rosservice type /clear
```

You will get something like:

```
std_srvs/Empty
```

This service is empty, this means when the service call is made it takes no arguments (i.e. it sends no data when making a **request** and receives no data when receiving a **response**). Let's call this service using `rosservice call`:

1.4.3 Command `rosservice call`

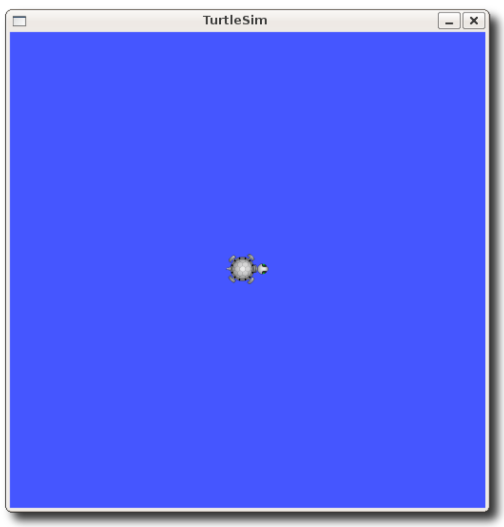
The command can be used as follows:

```
rosservice call [service] [arguments]
```

Here we'll call with no arguments because the service is of type empty:

```
rosservice call /clear
```

This does what we expect, it clears the background of the `turtlesim_node`.



Let's look at the case where the service has arguments by looking at the information for the service spawn:

```
rosservice type /spawn | rossrv show
```

The previous command does two things at once. First, it finds the message type of the service `/spawn` using `rosservice type [service]` command. Then, it shows the message content using the command `rossrv show`.

You will get something like:

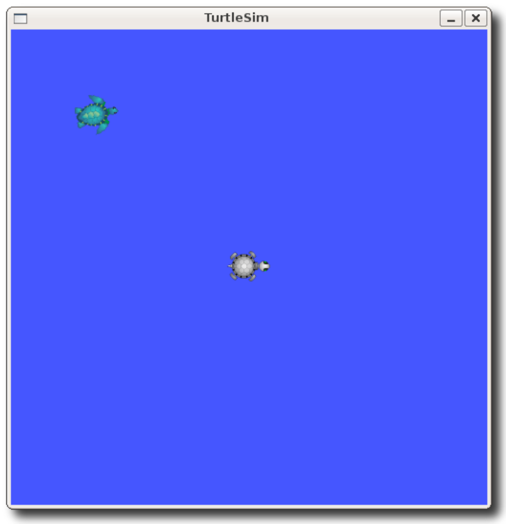
```
float32 x
float32 y
float32 theta
string name
---
string name
```

This service `/spawn` lets us spawn a new turtle at a given location and orientation. The name field is optional, so let's not give our new turtle a name and let turtlesim create one for us.

```
rosservice call /spawn 2 2 0.2 ""
```

Hint: You can use the autocomplete feature to get the service msg *fields* when you use `rosservice call [service] [args]` so you don't have to remember the `[args]` yourself. To do that, just hit `TAB` key twice after you write `rosservice call [service]`

After executing the previous command, you will get something like:



Until now, you have called services from the command line. There are three more things that you need to know.

- Writing a code for ROS service to execute certain functionality
- Writing ROS node that calls a service
- Writing custom service message

For writing ROS services and clients, I refer you to the following ROS WiKi page for more details.

For writing custom messages, I refer you to the following ROS WiKi page for more details.

1.5 ROSBag

The ROSBag is a powerful tool for you to record and playback data from ROS environment for future debugging and analysis. To start recording all topics available, simply type the following command in a separate terminal.

```
rosbag record -a
```

If you want to record to bag with specified name and specific topics (`/odom` and `/altitude` in this case), run the following command.

```
rosbag record -O file_name.bag /odom /altitude # O stands for Output name
```

You can record all topics subscribed to a specific node, split bag files, specify duration, and many more. Check the official ROS documentation webpage.

`rqt_bag` provides a GUI plugin for displaying and replaying ROS bag files.

1.6 Useful Video Tutorials

- ROS: Introduction, Installing ROS, and running the Turtlebot simulator
- Publishers and subscribers
- Python walkthrough of publisher/subscriber lab
- To learn more about Nodes and Topics, check the following video

1.7 Solutions

Solutions are available at Risc Github page.

1.8 Contributors

Mohamed Abdelkader.

CHAPTER 2

ROS Navigation

2.1 Introduction

In this tutorial you will learn how to program a mobile robot using ROS to navigate from point A to point B autonomously while avoiding obstacles.

You will learn by doing. Specifically, in each of the three main topics in this tutorial, you will follow the following steps.

- **Running a complete example.** First, for each topic, you will run a set of available ROS programs to perform the tasks discussed in this tutorial in order to get familiar with the tools.
- **Analyze examples**. Next, we will dig deeper into the examples and understand how it works.
- **Do exercises**. To make sure that you understand and enforce the concepts you learned, a set of exercises are provided for you to solve. Solutions for those exercises are available. **However**,

Important: You will only learn the concepts mentioned here by **practicing**. Remember to practice, practice, and practice . . . .

In this tutorial, you will work with a simulated robot called `TurtleBot` in the Gazebo simulator.

2.2 Prerequisites

This tutorial assumes the following.

- You are familiar with ROS basics e.g. topics, services, actions, how to write ROS nodes in Python, and ROS command line tools
- ROS Kinetic (Desktop-Full install) is installed on Ubuntu 16.04 LTS http://wiki.ros.org/kinetic/Installation/Ubuntu
- Gazebo 7. Comes by default with ROS Kinetic

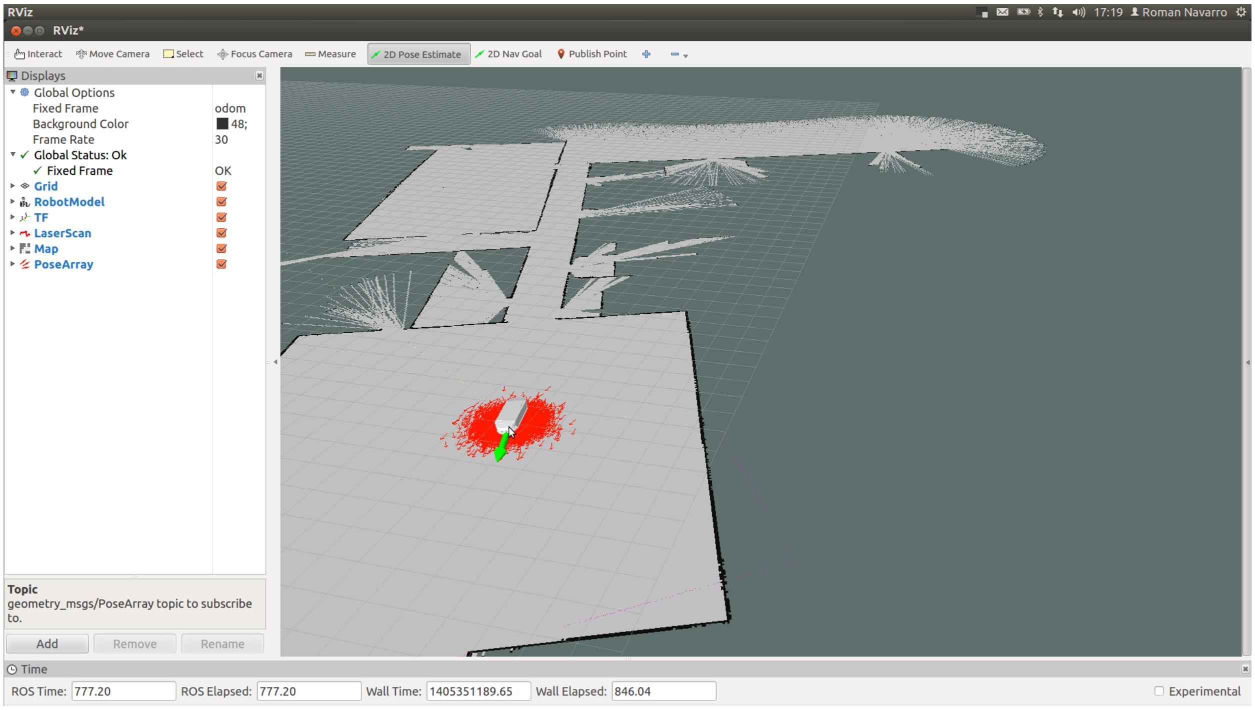
- A good PC. Recommended i5 with minimum of 8GB RAM
- Basic programing in Python

2.3 Topics Covered

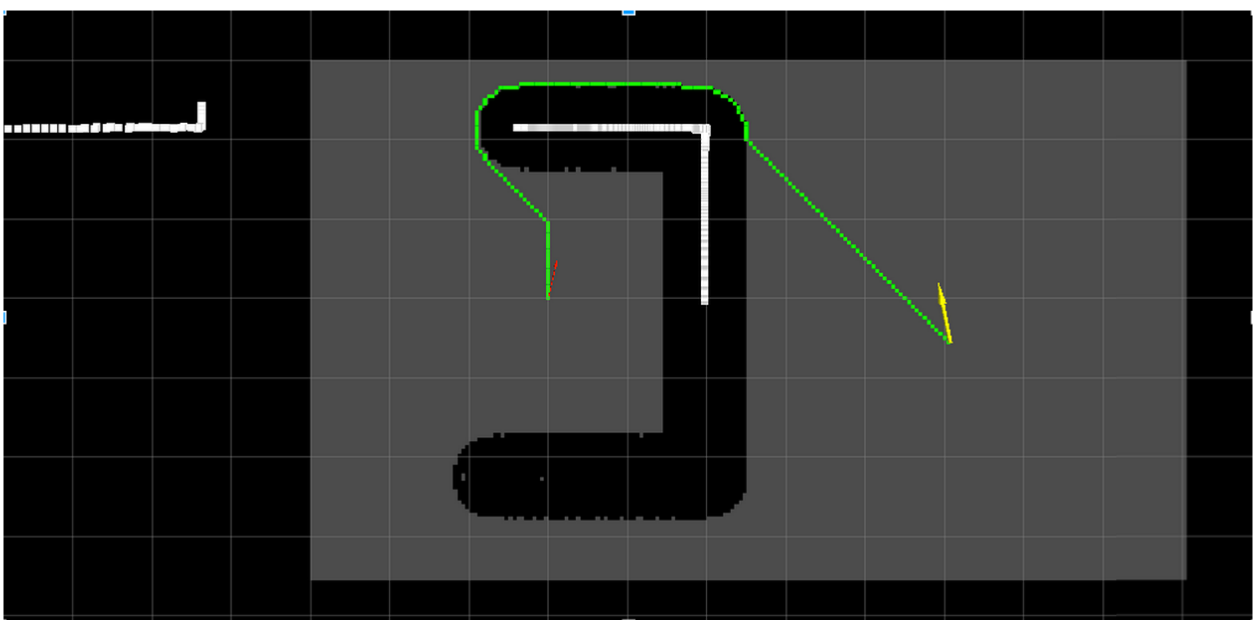
For a robot to navigate autonomously it needs the following.

- A **map** of the world or the environment of interest. You will learn how to create a map using laser scans and 2D SLAM ROS programs that are already available. You will also know how use a map that is already available. The ROS package that will be used for mapping is called `gmapping`

- **Localization**. A robot needs to know where it is inside the map in order to know how to go to a goal location. You will learn how to use a localization algorithm already implemented in ROS to help the robot estimate its location in a given map based on 2D laser scans. The ROS package that will be used for localization is called *amcl*, Adaptive Monte Carlo Localization.



- **Path planing**. This is the process of generating a sequence of points (path) between a start point and a goal point.
- **Path following**. This is the process of following the path that is planned while avoiding obstacles. The ROS package that will be used here for navigation is called *move_base*



In all the tutorials, you will be using `Rviz` which is a very powerful ROS tool for visualizing the status of your robot, sensor information, map building, navigation, and debugging.

2.4 Environment Setup

During this tutorial, you will work with a simulated robot called TurtleBot, to apply the concepts of navigation using ROS. The following image is a picture of the robot you will work with. It is a differential drive robot, that has a Kinect sensor for environmental mapping, wheel encoders for pose estimation.

../_static/kobuki.jpg

2.4.1 Install TurtleBot packages

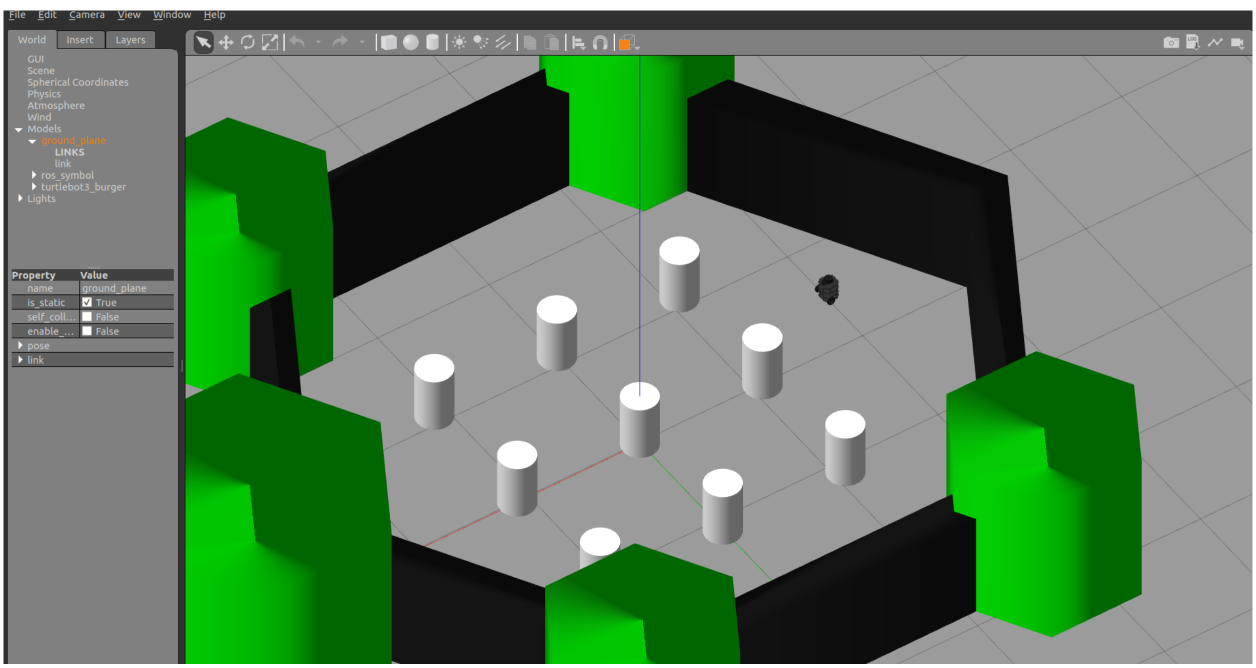
```
sudo apt-get install ros-kinetic-turtlebot ros-kinetic-turtlebot-apps ros-kinetic-turtlebot-interactions ros-kinetic-turtlebot-simulator ros-kinetic-turtlebot-gazebo -y
```

After installation is done, check that the simulation works in Gazebo. Execute the following commands in a shell terminal.

```
roslaunch turtlebot_gazebo turtlebot_world.launch
```

Attention: It might take long time if you are opening the previous Gazebo world for the first time. Just be patient.

You should get something similar to the following.



2.5 Tele-operating the Robot

You will need to move the robot around somehow in order to build a map of the world in the coming sections. You can move it using a *keyboard* or a *joystick*.

`turtlebot_teleop` package provides nodes and launch file to move the robot by either a *keyboard* or a *joystick*. There is one *launch* file for keyboard teleoperation and three *launch* files for three different joysticks. To navigate to the launch file folder, execute the following.

```
roscd turtlebot_teleop/launch
```

To move the robot using a keyboard, execute the corresponding *launch* file in a separate terminal, after you launch the TurtleBot's world.

```
roslaunch turtlebot_teleop keyboard_teleop.launch
```

Use the keys as mentioned on the screen to move the robot.

To move the robot using a joystick (we will assume Logitech F710 joystick), execute the following.

```
roslaunch turtlebot_teleop logitech.launch
```

Hint: You you will need to press certain button combination in order to control the robot with the joystick. Read the instruction in the `logitech.launch` file.

Important: Make sure that your joystick is given the required privileges. Use `sudo chmod a+rw /dev/input/jsX` (`X` is the device number) to give the required privileges to your joystick.

2.6 Rviz

Rviz (ROS visualization) is a 3D visualizer for displaying sensor data and state information from ROS. You can also display live representations of sensor values coming over ROS Topics including camera data, infrared distance measurements, sonar data, and more.

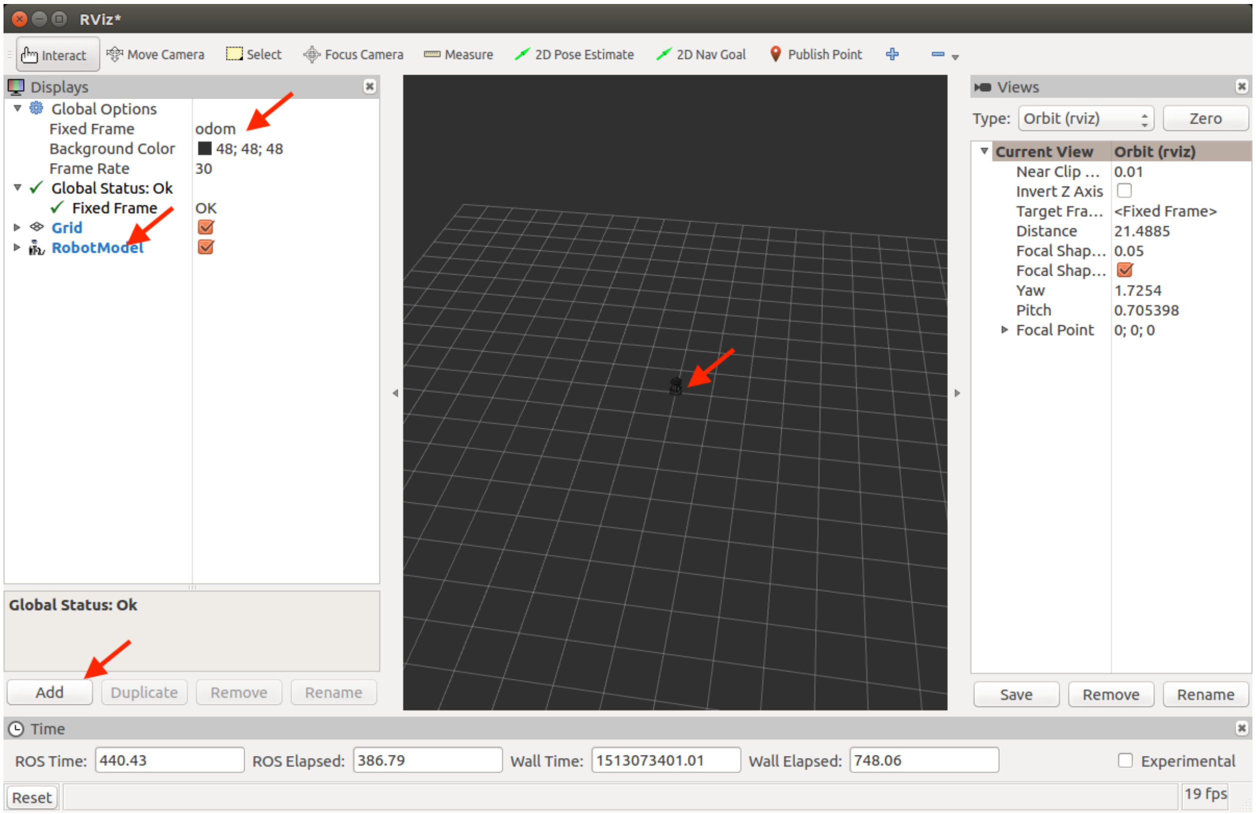
We will be using Rviz all the way in this tutorial. Now, let's see how we can show simple things in Rviz.

Running Rviz. Make sure that you launched a turtlebot world. Next, in a separate terminal, run rviz using the following command.

```
rosrun rviz rviz
```

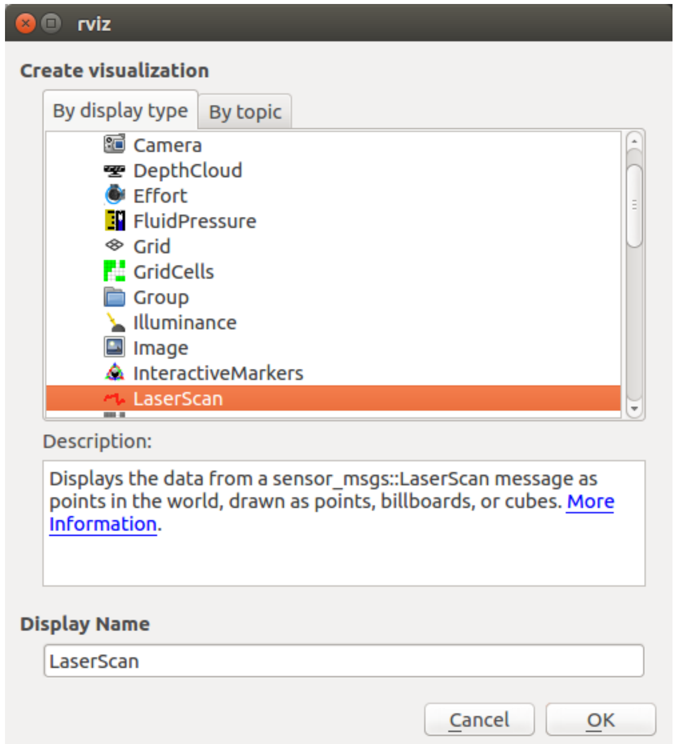
Adding displays. Next, we will need to add the information that we would like to visualize. This is called *Displays*. On the left side of Rviz, there is a column called *Displays*. The first thing we need to adjust is the **Fixed frame** field. Choose the `odom` frame. This is the frame that is created at the initial point of the robot when you launched your turtlebot world, then it becomes fixed for the rest of the simulation. It's called local fixed frame.

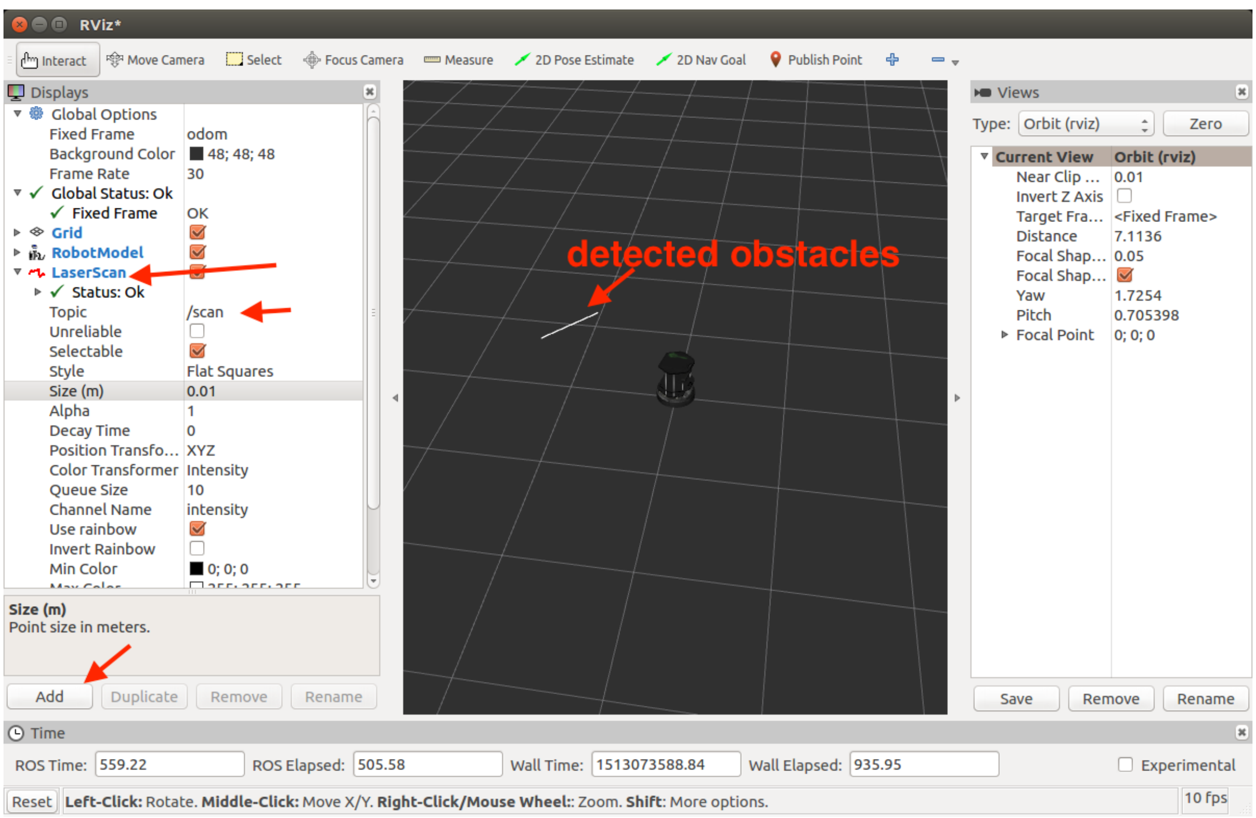
Adding Robot Model. To show the robot 3D model, we will need to add a display for that. Click on the **Add** button in the lower left corner of the *Displays* column. Then, choose **RobotModel**. You should see the robot model in the middle screen.



Adding a display for laser scans. To show what the laser scanner detects on the robots, you can add a `LaserScan` display. After adding the display, you will need to specify the topic that has the laser scans reading. In this case it is called `/scan`

See following snapshots to know what to expect your rviz configs to be like.





Now, you have a basic idea on how to use Rviz to visualize your robots states. Later, we will also use it to visualize the map we built or while we are building it, paths we want to navigate, and how to use it to set goal waypoints.

Hint: If you close Rviz, you will lose the displays and the configs you made. You can save the current configs you did in order to load it quickly next time you launch Rviz. Just use the *File* menu and choose *save config as*.

Now it's time to build a map!

2.7 Mapping

The first step we need to do in order to be able to perform autonomous navigation is to **build a map**.

In this tutorial we will learn how to create a 2D map with a ROS package called `gmapping`. Here is the definition of the package according to the official WiKi (http://wiki.ros.org/gmapping)

Note: The gmapping package provides laser-based SLAM (Simultaneous Localization and Mapping), as a ROS node called slam_gmapping. Using slam_gmapping, you can create a 2-D occupancy grid map (like a building floorplan) from laser and pose data collected by a mobile robot.

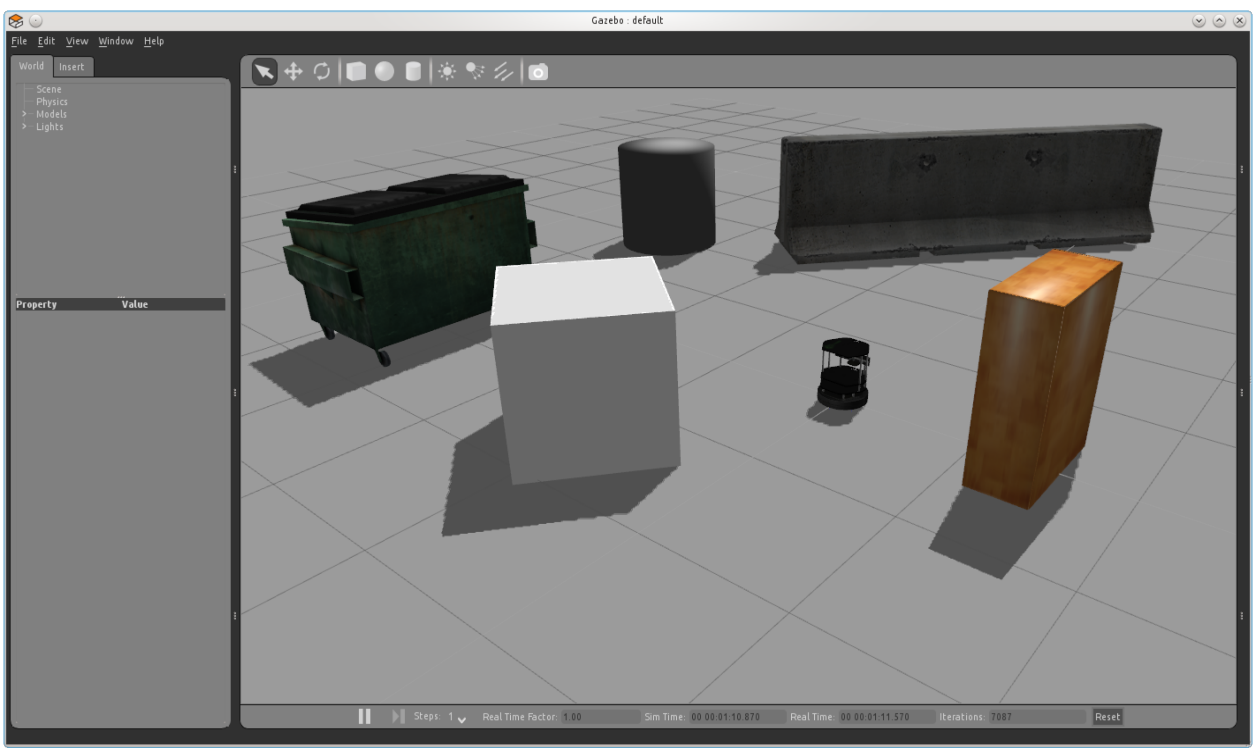
Although there are other packages that allow to build **3D** maps, but we will only stick to 2D mapping in this tutorial.

So basically, we will be performing 2D SLAM in order to construct a 2D map of a certain environment. To do that, as mentioned, we will use `gmapping` package. This package takes *laser scans* and *robot odometry* and outputs a map expressed as *occupancy grid*.

To start, we will see how to use mapping through an example. First let's bring up our Gazebo world.

```
roslaunch turtlebot_gazebo turtlebot_world.launch # you can launch different world by adding world_file:=worlds/willowgarage.world
```

The playground world with a TurtleBot2 looks like this:



To start building a map, let's run the gmapping node

```
roslaunch turtlebot_gazebo gmapping_demo.launch
```

Next, run **Rviz** in order to visualize the map you build in real-time.

```
rosrun rviz rviz
```

Add the following displays, in order to visualize the robot, laser scans, and the map

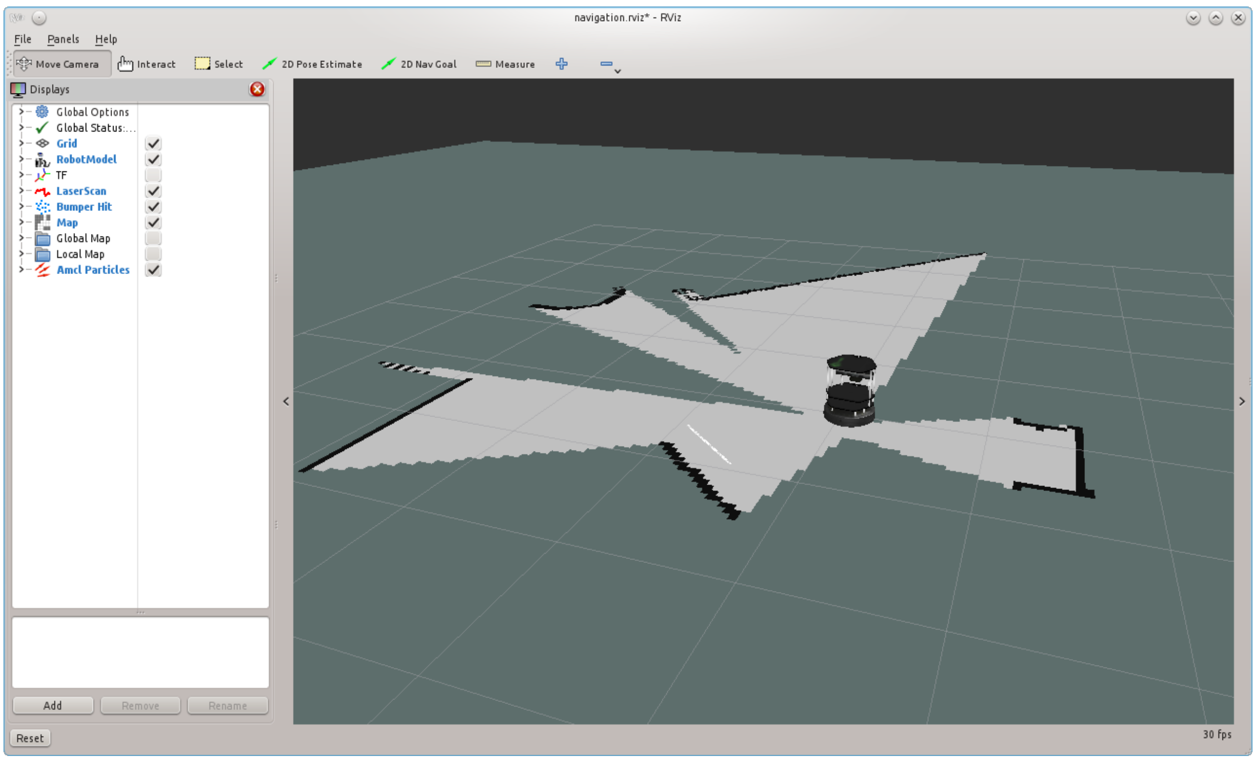
- `RobotModel`, `LaserScan` with topic name /scan, `Map` with topic name /map

Use your favorite teleoperation tool to drive the TurtleBot around the world, until you get satisfied with your map. The following capture shows the mapping process after turning 360 degrees.

For example, to use the keyboard to drive the robot, launch the corresponding launch file as you did before in the *Tele-operating the Robot* section.

```
roslaunch turtlebot_teleop keyboard_teleop.launch
```

Start driving the robot using keyboard keys and observe how the map is updated in **Rviz**.



Once you get satisfied about your map, you can save it for later use. To save the map execute the following command inside the folder you would like to save the map inside.

```
rosrun map_server map_saver -f <your map name>
```

Your saved map is represented by two files.

- YAML file which contains descriptions about your map setup
- Grayscale image that represents your occupancy grid map, which actually can be edited by an image editor

Open `gmapping_demo.launch` and see what it does.

2.8 Localization

After we build the map, we need to localize the robot on that map. In order to perform proper navigation, a robot needs to know in which position of the map it is located and with which orientation at every moment.

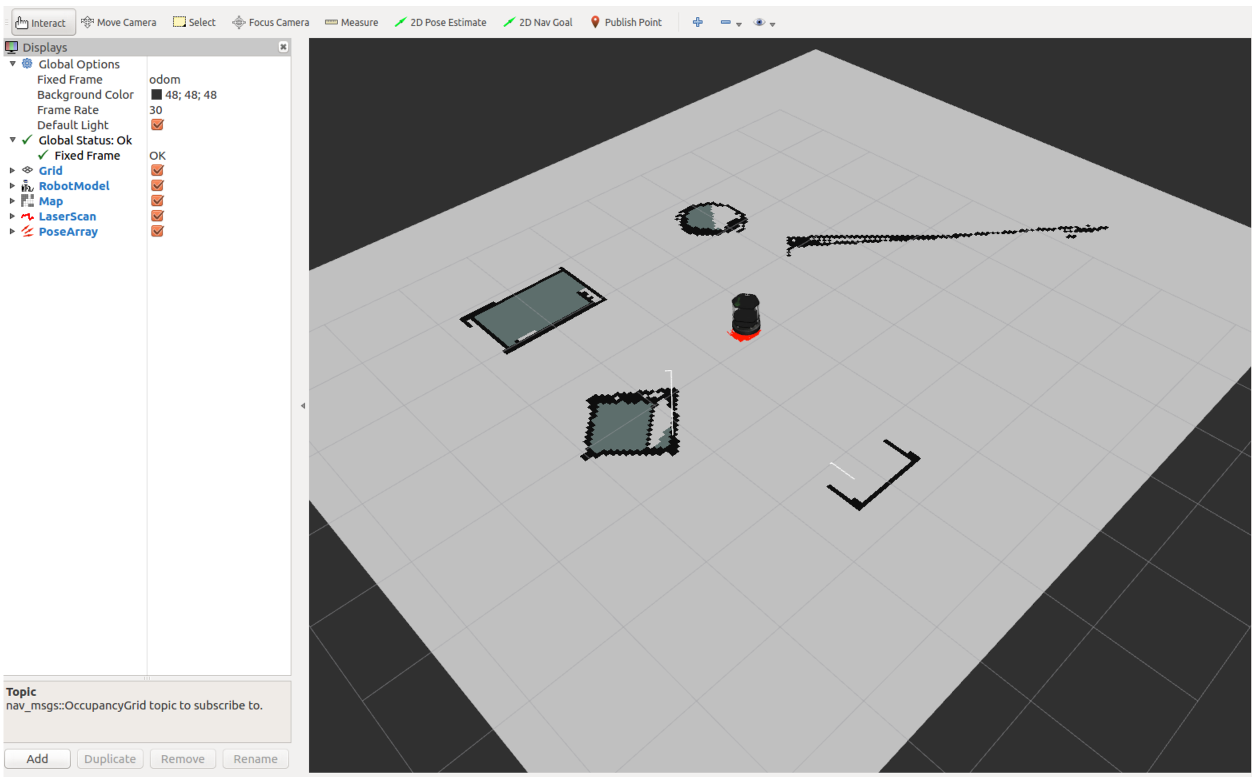
Run the the following commands

```
roslaunch turtlebot_gazebo turtlebot_world.launch # brings turtlebot in simulation
roslaunch turtlebot_gazebo amcl_demo.launch # starts amcl node, add *map_file:=<full path to your map>*, to use saved map from previous seciton
roslaunch turtlebot_teleop keyboard_teleop.launch
rosrun rviz rviz
```

Add the following displays, in order to visualize the robot, position and orientation.

- `RobotModel`
- `PoseArray` topic name /particlecloud

Move the robot with the keyboard and see in rviz how things are changing. So amcl node is a probabilistic localization system for a robot moving in 2D. It implements the adaptive (or KLD-sampling) Monte Carlo localization approach, which uses a particle filter to track the pose of a robot against a *known map*.



Also analyze `amcl_demo.launch` file and check what it does.

Hint: You can use predefined Rviz configuration by running the following command

```
roslaunch turtlebot_rviz_launchers view_navigation.launch
```

2.9 Path Planning/Following (NOT COMPLETED)

Let's run the *acml_node* from previous section

```
roslaunch turtlebot_gazebo amcl_demo.launch

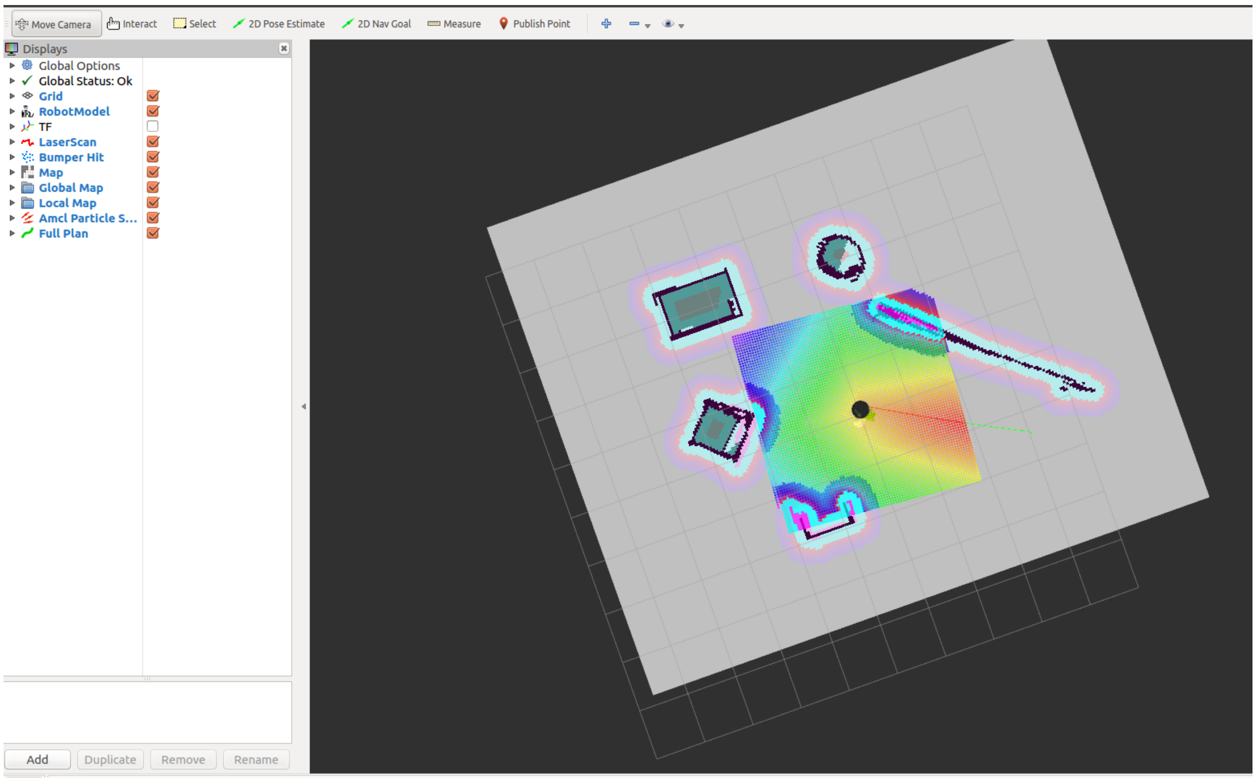
rostopic list
```

Make 2D nav goal from rviz

```
rostopic echo /move_base_simple/goal
```

Let's see how topic has been changed.

Which mean we can publish to this topic as well.



2.10 References

https://www.youtube.com/playlist?list=PLK0b4e05LnzZA_fWYi1_VEuBzNw9BGo6s

2.11 Contributors

Mohamed Abdelkader and Kuat Telegenov.

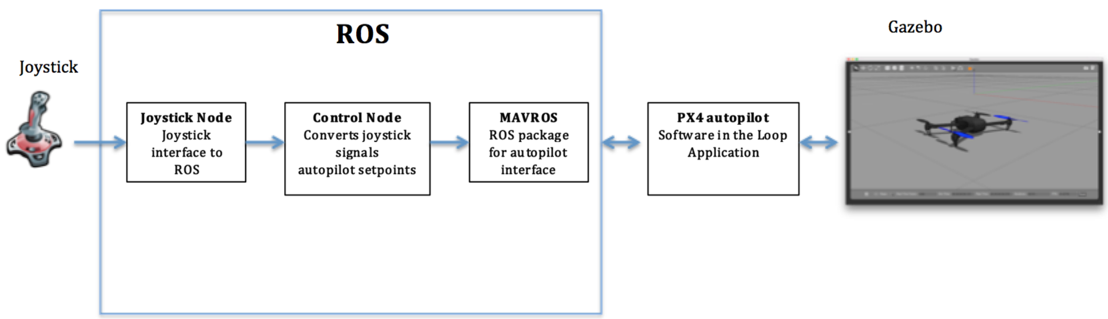
CHAPTER 3

3D Modeling

CHAPTER 4

Software in the Loop Joystick Flight

This tutorial explains the steps required to fly a simulated quadcopter in the Gazebo simulator using a real joystick. The following diagram shows how the system components work together.



4.1 Hardware Requirements

- Desktop Linux Machine with minimum of 8GB RAM, 16GB recommended, Ubuntu 20.04 installed
- Joystick

4.2 Software Requirements

- **Ubuntu 20.04**
- **ROS Noetic** (full desktop installation)
- **Gazebo**: will be automatically installed with ROS
- **PX4 firmware** installation on Linux: Autopilot software which includes the software-in-the-loop firmware
- **MAVROS** package: Autopilot ROS interface
- **Joy** package: Joystick ROS interface

Note: In this tutorial, it is assumed that the reader is familiar with basic Linux commands, ROS Basics.

4.3 Setup Steps

- Install `QGroundControl` from here. Use the AppImage option.

4.4 PX4 SITL with Gazebo System setup

Follow the steps from the video. This will install PX4 firmware and it's dependencies on your computer and build it.

https://youtu.be/9Mb-aV3lmZ0

4.5 Launching Gazebo with ROS Wrappers

Now, you are ready to launch Gazebo + PX4 SITL app + ROS + MAVROS. To do that, execute the following command.

```
roslaunch px4 mavros_posix_sitl.launch
```

You should be able to see `/mavros` topics using `rostopic list` in a new terminal. Also if you execute `rosnode list` in a new terminal, you should see following

```
$ rosnode list
/gazebo
/gazebo_gui
/mavros
/rosout
```

To double check that MAVROS node is connected properly to the PX4 SITL app, try to `echo` some topics _e.g._

```
rostopic echo /mavros/state
```

Which will show if the mavros node is connected to the PX4 SITL app or not.

Now, you can monitor the drone's states and control it via a MAVROS node.

- As mentioned, in this tutorial, we are going to learn one basic way of controlling the quadcopter's position via a joystick.
- There is a flight mode in PX4 autopilot which is called **OFFBOARD** mode. This mode allows the autopilot to accept specific external commands such as position, velocity, and attitude setpoints. You cannot mix between different setpoints _e.g._ velocity setpoints in x/y and position in z.
- A MAVROS node provides setpoint plugins which will listen to a user input on specific setpoint topics. Once the user publishes to those specific setpoint topics, the mavros node will transfer those setpoints to the autopilot to execute.
- If the autopilot's flight mode is **OFFBOARD**, the autopilot will accept the received setpoints and execute them.
- We will send position setpoints to the autopilot via a setpoint topic that is available in MAVROS. Once set points are received in that topic, the mavros node will send it to the autopilot.
- The setpoint topic that we will use in this tutorial is `/mavros/setpoint_raw/local`. This topic accepts both position and velocity setpoints according to a specific flag. Next, we will create our custom simple ROS package in which we create a simple ROS node that listens to joystic commands from a ROS topic. Then, it will convert joystic commands to position setpoints which will be published to the `/mavros/setpoint_raw/local` topic. Finally, MAVROS will take the position set points and send them to the autopilot.

You might be asking, how are we going to get the joystick commands? The next section explains that.

4.6 Joystick Package Installation & Usage

A package named `joy` is going to be used to interface a joystick to ROS. To install that package, simply execute the following command in the terminal.

```
sudo apt-get install ros-kinetic-joy
```

You will need to setup permissions before you can use your joystick.

- Plug a joystick
- Check if Linux recognizes your joystick

```
ls /dev/input/
```

You will get an output similar to the following.

```
by-id    event0  event2  event4  event6  event8  mouse0  mouse2  uinput
by-path  event1  event3  event5  event7  js0     mice    mouse1
```

As you can see, the joystick device is referred to as `jsX` where `X` is the number of the joystick device.

Let's make the joystick accessible to the joy ROS node.

```
ls -l /dev/input/jsX
```

You will see something similar to:

```
crw-rw-XX- 1 root dialout 188, 0 2009-08-14 12:04 /dev/input/jsX
```

If XX is `rw`: the js device is configured properly. If XX is `--`: the js device is not configured properly and you need to:

```
sudo chmod a+rw /dev/input/jsX
```

Test the `joy` node. First, start `roscore` in a terminal. In another terminal,

```
# set the joystick device address
rosparam set joy_node/dev "/dev/input/js0"
# run the joy node
rosrun joy joy_node
```

In another terminal, echo the `joy` topic and move the joystick to see the topic changes

```
rostopic echo /joy
```

You should see an output similar to the following.

```
header:
seq: 699
stamp:
  secs: 1505985329
  nsecs: 399636113
frame_id: ''
axes: [-0.0, -0.0, -0.8263657689094543]
buttons: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Now, let's write a custom node that reads joystick's commands and convert them to position setpoints to control the quadcopter's poisiton in Gazebo.

4.7 Custom Setpoint Node

Now, it's time for some coding! You will write a ROS node in Python that listens to the `/joy` topic that is published by the `joy` node, and converts the joystick commands to xyz position setpoints. Then, it will publish the calculated position setpoints into `/mavros/setpoint_raw/local`

Publishing to `/mavros/setpoint_raw/local` topic is not enough to get the autopilot to track the setpoints. It has to be in **OFFBOARD** mode. So, in your custom node, you will have to send a signal to activate this mode, only once. You need to **remember** that for this mode to work, you will need to be publishing setpoints beforehand, then, activate it, and continue publishing setpoints. **If you don't publish setpoints at more than 2Hz, it will go into a failsafe mode**.

First, create your custom ROS package. The code is commented so you can get an idea of what each part does. Go through code and try to understand it! The python script and launch file are not complete, so you need to add some lines of code where needed.

```
cd ~/catkin_ws/src
catkin_create_pkg mypackage std_msgs mavros_msgs roscpp rospy
cd mypackage
# usually python scripts (nodes) are placed in a folder called scripts
mkdir scripts
cd scripts
wget https://raw.githubusercontent.com/risckaust/risc-documentations/master/src/
↪gazebo-flight/setpoints_node.py
```

Make the python file an executable,

```
chmod +x setpoints_node.py
```

Make a **launch** folder. We will create a ROS launch file to run everything at once. Open the launch file and understand what every line executes.

```
cd ~/catkin_ws/src/mypackage
mkdir launch
cd launch
wget https://raw.githubusercontent.com/risckaust/risc-documentations/master/src/
↪gazebo-flight/joystick_flight.launch
```

In a fresh terminal, you can run the launch file by executing

```
roslaunch mypackage joystick_flight.launch
```

Now, you should see a quadcopter in Gazebo flying at a fixed height and responding to your joystick commands.

> **Warning:** Always make sure that you have joystick permissions configured properly.

Mohamed Abdelkader.

CHAPTER 5

Quadcopter Assembly

5.1 Basic principles

Feel free to place the components anywhere on the frame but take care of wires. Refer to quadcopters we already have in the lab. Carefully choose zipties, shrinking tubes, double sided tapes or soldering for different situations. Generally, for fixing motor wires we use zipties. Shrinking tubes are for permanent connection between wires when soldering.

5.2 Preliminaries

This tutorial assumes you have the following skills:

- *ROS Basics*.
- Soldering, if not, please refer to basic skill video.
- Basic knowledge about LiPo batteries. Answer the following questions. You may read this article.
  - What do 3s, 4s mean?
  - What does 20c mean?
  - What does 1400mAh mean?
  - What are the parameters of your battery?
  - How to charge LiPo battery? How to measure it voltage using battery meter?
  - What's the minimum voltage to use a LiPo on the quadcopter?
- Basic knowledge about motors. Answer the following questions. You may refer to this article.
  - Different types of motors. We are using brushless motor for quadcopters.
  - What does KV2200 means? What will be changed if KV number grows?
  - What are the parameters of your motors?

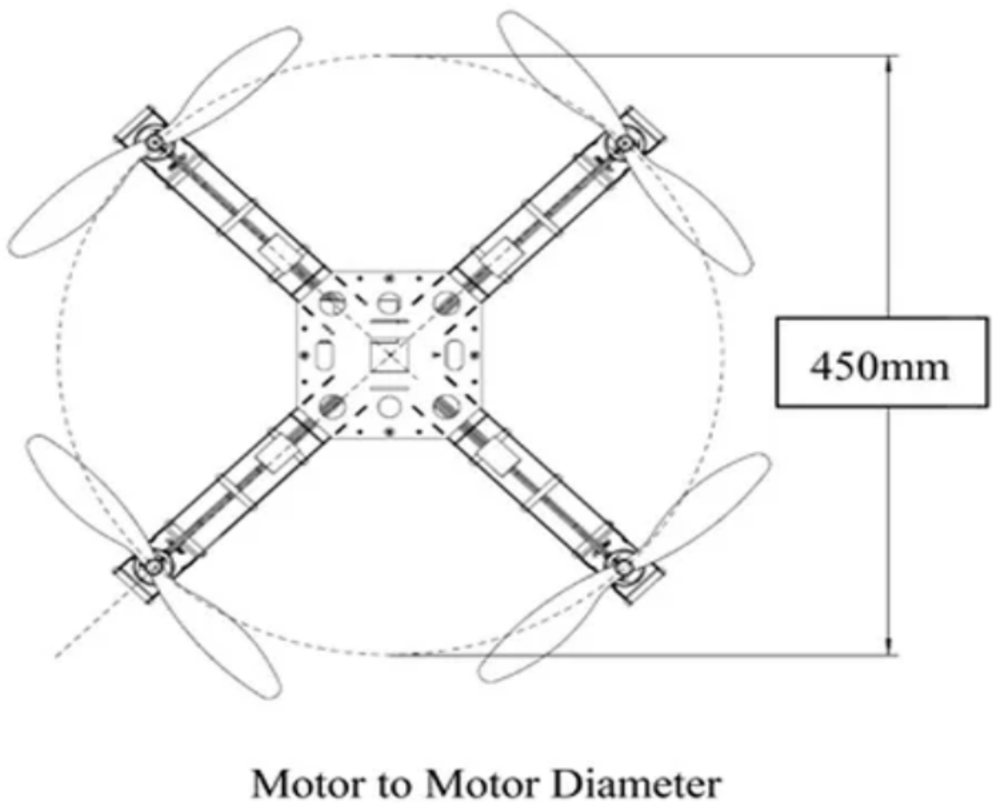
Danger: Do not leave your battery plugged in your quadcopter for a long time and never discharge a LiPo battery below 3.4V per cell.

5.3 Hardware assembly

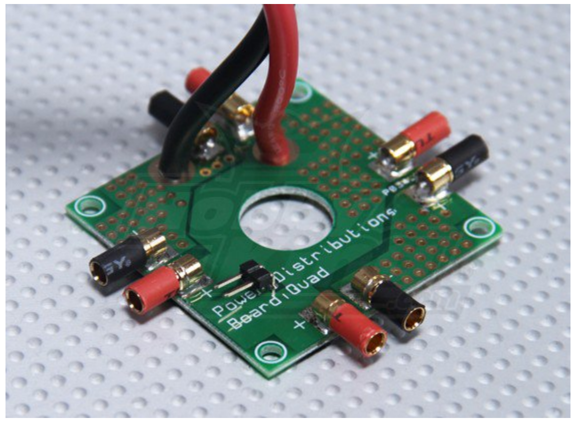
5.3.1 Introduction

You will need

- Quadcopter frame. Frames 250 or 330 will be a good start. The value 250/330 means the motor to motor diameter, as shown below.



- Power distribution board to distribute power from a battery to 4 ESCs (no need in case of F330 and F450).



- Flight Controller. Use any flight controller available in the lab. Just make sure you have compatible power modules, receivers, GPS, and other additional modules. The documentations for each board are available here.
- Brushless motors and propellers. For mini quad pilots, 3-blade (or tri-blade) propellers are equally popular as the two blades, they are commonly used in both racing and free-style flying. Some people prefer tri-blades because it has more grip in the air. Basically, by adding more blade it's effectively adding more surface area, and therefore it generates more thrust in the expense of higher current draw and more drag.

Note: There are 2 types of format that manufacturers use.

L x P x B or LLPP x B where L- length, P – pitch, B – number of blades.

For example 6×4.5 (also known as 6045) propellers are 6 inch long and has a pitch of 4.5 inch. Another example, 5x4x3 (sometimes 5040×3) is a 3-blade 5 propeller that has a pitch of 4 inch. "BN" indicates Bullnose props.

Sometimes you might see **R** or **C** after the size numbers, such as 5x3R. **R** indicates the rotation of the propeller, which stands for "reversed". It should be mounted on a motor that spins clockwise. **C** is the opposite, should be used with motors that spins counter-clockwise.

- Electronic speed controller (ESC) controls and regulates the speed of an electric brushless motor. All ESCs comes with a rating. The Turnigy Multistar ESC shown below has a rating of 10A, meaning it can draw a maximum continuous current of 10A. Anything higher than 10A will eventually burn or damage the ESC.

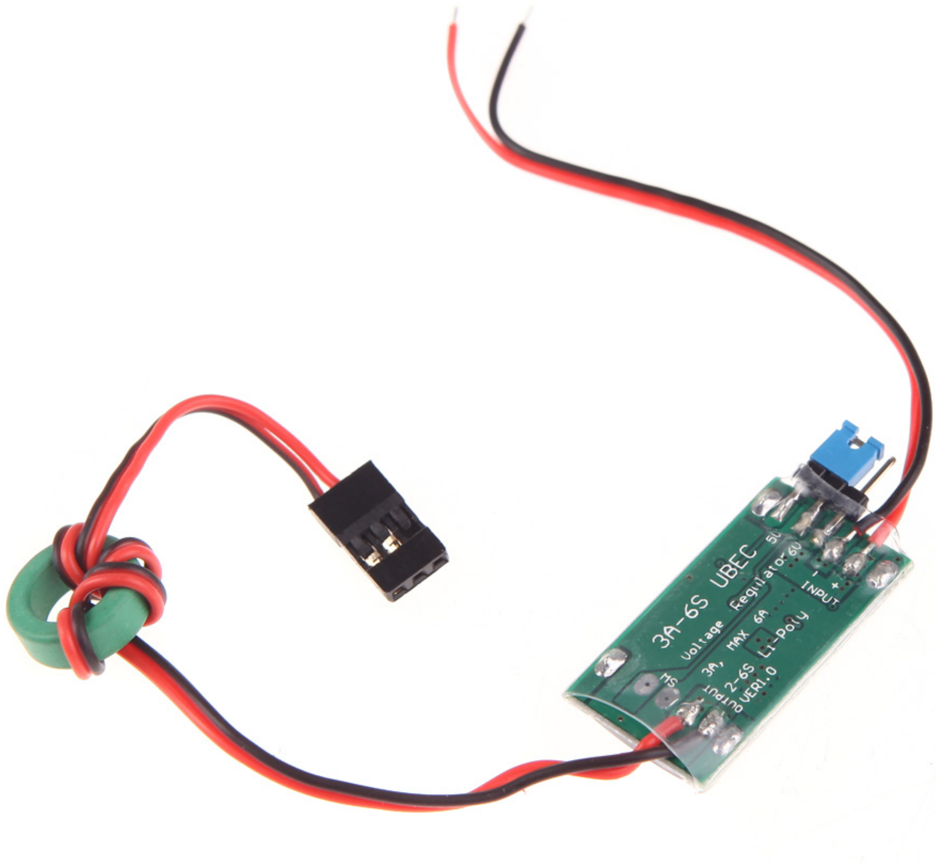
Note: Drawing 10A for a long time (~10mins) will heat up the ESC and damage it as well. Always use a higher rating ESC for your setup. E.g. If your motor draws 10A (at full throttle), use either a 12A or a 15A. If the 12A and the 15A

ESC weight approximately the same, choose the 15A. A higher rating ESC will prevent overheating. To handle more power, a high rating ESC will be required. As the rating goes up, the weight, size and cost of the ESC go up as well. Always consider how much power you will need by looking up your motor specification (Max current motor drawn).

- Remote control system. A remote control (RC) radio system is required if you want to manually control your vehicle. In addition to the transmitter/receiver pairs being compatible, the receiver must also be compatible with PX4 and the flight controller hardware.

It's recommended to use Taranis X9D Plus transmitter with X8R receiver as shown below

- UBEC (Universal Battery eliminator circuit) to convert voltage to power Odroid. A BEC is basically a step down voltage regulator. It will take your main battery voltage (e.g. 11.1 Volts) and reduce it down to ~5 Volts to safely power your Odroid and other electronics.

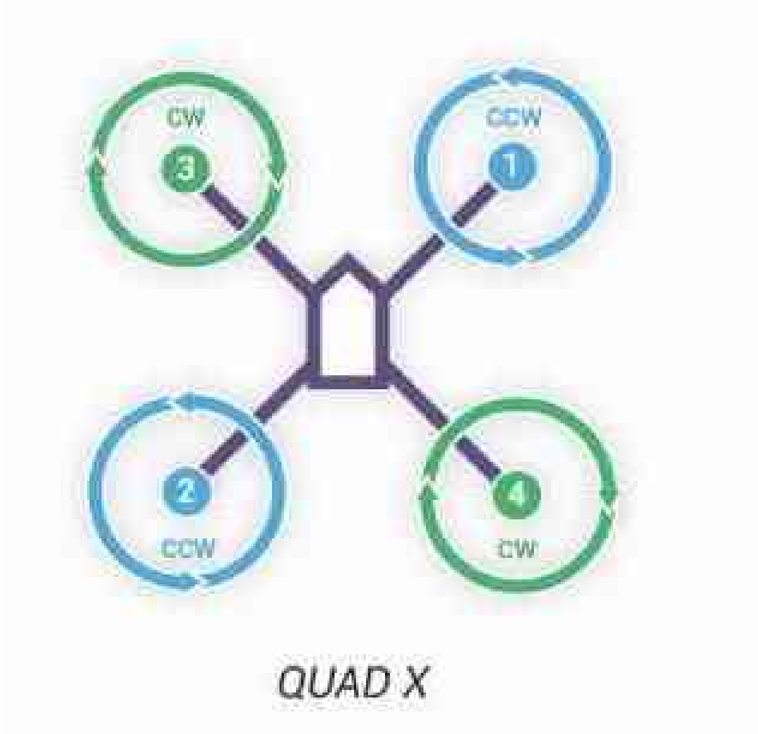


- Power module. It is the best way to provide power for flight controller unit. It has voltage and current sensors that allows autopilot to estimate remaining battery charge precisely. Usually it comes with every autopilot controller as a default kit. Check official documentations to match right power module to a selected flight controller.

- LiPo battery. Assuming you know what is the balancer, cell count and voltage, capacity and C-rating.

5.3.2 Assembly process

- Assemble the frame. Attach the power distribution board to it (no need if you use frame with soldered pads).
- Mount the motors to the frame. Mind CW and CCW directions. They should be mounted as follows. We usually use **X** configuration.



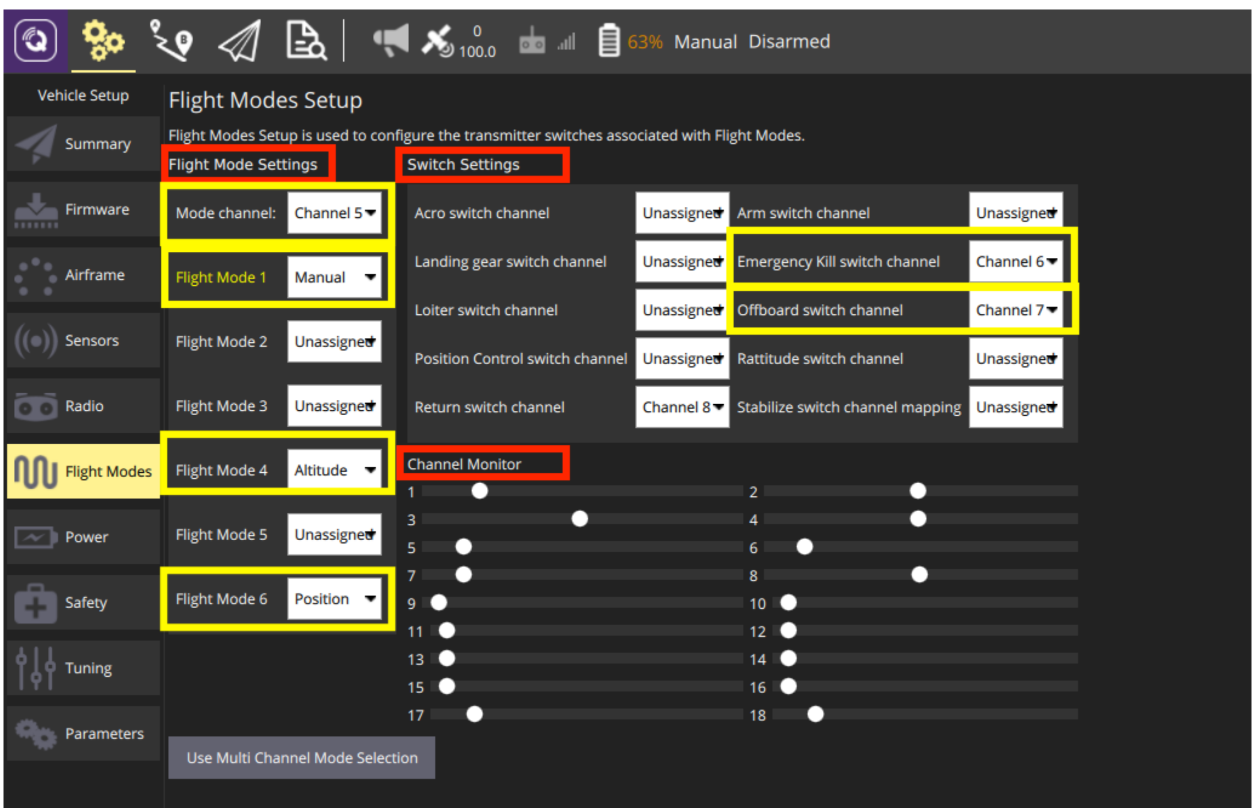
Important: Do not install propellers now.

- Connect ESCs to motors and plug ESCs to power distribution board (or solder them to the frame). As for now, connect motors to ESCs arbitrary, later you will set them properly by switching any two wires.
- Install power module on the frame. One end should be plugged to power distribution board (or soldered to the frame) and the other end to the battery. DON'T plug it to the battery for now.
- Install flight controller on the frame. Take a look at your flight controller and make sure the arrow is pointing to the front between motor 1 and 3. To mount the controller to the frame, use thick double side tape to damp the vibrations.
- Plug cable from power module to `POWER` port of your flight controller.
- Plug buzzer and switch to their corresponding ports on flight controller.
- Connect each of your ESCs servo cables to the corresponding **MAIN OUT** output, eg. motor 1 to **MAIN OUT** port 1.
- Binding process for FrSky X8R
  - Connect the RCIN port from Pixhawk to SBUS port on X8R
  - Turn on the X8R while holding the **F/S** button on the module. Release the button.
  - Press the **Menu** button on your Taranis X9D
  - Go to page 2 by pressing **Page** button.
  - Scroll down with **-** button until you see **Internal RF** line.
  - Select **[Bind]** line, and press **ENT** button. The RED LED on the X8R receiver will flash, indicating the binding process is completed
- For this stage there's no need to install Odroid.

5.4 Calibration process

- Download `QGroundControl` on your computer and open it.
- Install Stable PX4 firmware.
- Set the airframe, for example: Generic 250 Frame, Flamewheel F330 or Flamewheel F450 depending on your frame. Follow steps from this page.
- Calibrate Compass, Accelerometer, and Level Horizon.
- Calibrate the Radio.
- In `Flight Modes` tab under the **Flight Mode Settings** and **Switch settings** sections set:
  - **Mode Channel** to SB (SB switch labeled on your Taranis X9D)
  - **Mode 1: Manual**.
  - **Mode 4: Altitude**. Climb and drop are controlled to have a maximum rate.
  - **Mode 6: Position**. When sticks are released the vehicle will stop and hold position.
  - **Emergency Kill switch channel** to SF (SF switch labeled on your Taranis X9D). Immediately stops all motor outputs. The vehicle will crash, which may in some circumstances be more desirable than allowing it to continue flying.
  - **Offboard switch channel** to SA (SA switch labeled on your Taranis X9D).

You should have similar as shown in the picture below. Channels for **Flight Mode Settings** and **Switch Settings** might differ.



Hint: If you set everything right, you will see changes in **Flight Mode Settings** section highlighted as yellow. Also, moving sticks, dials and switches will be reported in **Channel Monitor** section.

- In `Power tab` write the parameters of your battery (Number of cells), calibrate the battery voltage and ESCs (if you use DJI ESCs, no need to calibrate them).
  - Press **Calculate** on the **Voltage divider** line
  - Measure the voltage with Digital Battery Capacity Checker by connecting it to the battery
  - Enter the the voltage value from the Digital Battery Capacity Checker and press **Calculate** button
  - To calibrate ESC press **Calibrate** under **ESC PWM Minimum and Maximum Calibration** and follow on-screen instructions
- Arm your quadcopter, and check if all motors are rotating in the direction intended. If no, switch any two wires that are connected to ESC. To arm the drone, put the throttle stick in the bottom right corner. This will start the motors on a quadcopter. To disarm, put the throttle stick in the bottom left corner.
- Now you can install propellers. Note that there are CW and CCW propellers as well.

Danger: After you install propellers, make sure to keep battery or receiver disconnected while you are working on your quadcopter. Someone may use transmitter bounded to your drone for their own quadcopter as well. The same transmitter can arm several quadcopters!

- Follow this guide to perform **PID** tuning for your quadcopter if necessary (no need for F330 and F450 frames).

5.5 Flying in manual mode

- Read First Flight Guidelines and Flying 101.
- Make sure you switch **Kill switch** to off. Select **Manual** as your flight mode.
- Check the battery level, make sure it's enough to perform your first flight.
- Put the quadcopter in the cage and arm. Slowly add throttle while keeping it in the middle of the cage by controlling pitch and yaw.

Important: Always check the battery before flying

5.6 Odroid installation

- Mount Odroid XU4 on the drone
- Solder the UBEC input cable to the power distribution board (or the frame)
- Solder Odroid DC Plug Cable to female servo cable and connect to the UBEC output cable
- In case of MindPX simply connect micro-USB cable to `USB/OBC` from the Odroid USB port. In case of Pixhawk use FTDI module. Use servo cable to solder three wires to GND, TX, and RX (refer to page 8 of the FTDI datasheet file). After that solder these three wires to corresponding **TELEM2** port cable. Note that GND connects to GND, RX to TX, and TX to RX.

- Plug in the DC power cable to the Odroid and check if it's powered from the battery

5.7 Troubleshooting

- Motors are not rotating while armed and rotates with higher throttle
  - Check `PWM_MAX` and `PWM_MIN` in parameters and make sure it's associated with ESCs
- Motor are not rotating or rotating partially.
  - Set `PWM_RATE` value to default.
- Drone goes high during take-off and hits the ceiling, even though after take-off the throttle stick is all they way down
  - Try to lower `MPC_THR_HOVER` value

5.8 Contributors

Yimeng Lu and Kuat Telegenov.

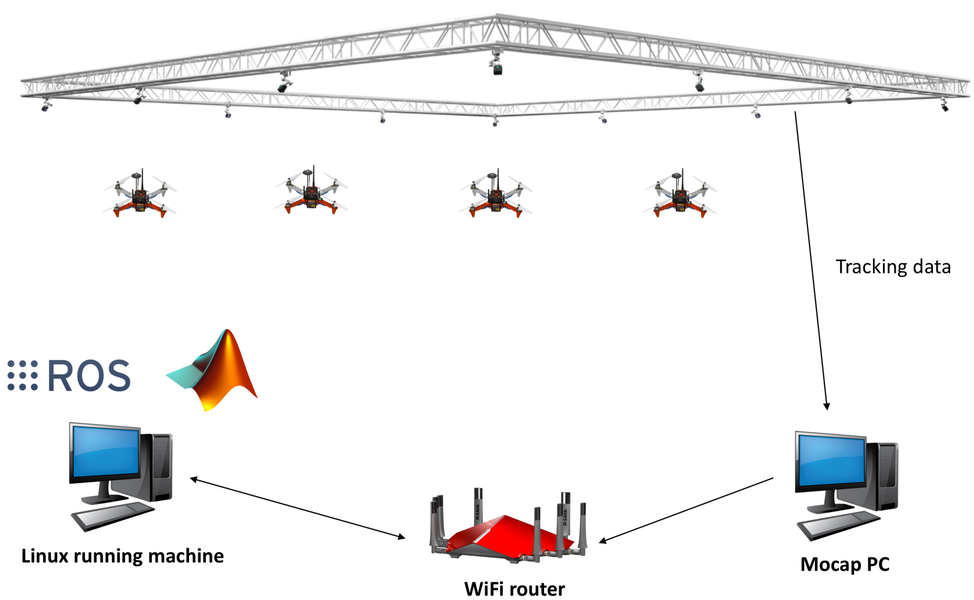
CHAPTER 6

Indoor flying

6.1 System Architecture

In order to start flying the quadcopter indoor, we need the position and orientation feedback for this.

This section will guide you how to use OptiTrack Motion Capture System, how to stream position and orientation data to ROS, and feed it to your flight controller. Finally you will be able to fly your drone inside the arena in position mode.



The overall systems has following main elements:

- OptiTrack Motion Capture System
- Object to be tracker, eg. quadcopter, ground vehicles.

- Controller

Let's discuss each element in details

6.1.1 Motion capture system

OptiTrack motion capture system (Mocap hereinafter) works as follows. The overhead cameras send out pulsed infrared light using the attached infrared LEDs, which will then be reflected by markers on the object and detected by the OptiTrack cameras. Knowing the position of those markers in perspective of several cameras, the actual 3D position of the markers in the room can be calculated using triangulation. Simply Mocap provides high precision indoor local position and orientation estimation. Position is meters and orientation is in quaternion, which can be converted Euler angles in radians. In RISC lab we use twenty **Prime17w** cameras that are installed in the flying arena.

All cameras are connected to a single Mocap PC through network switches. Motive Optical motion capture software is installed on this PC.

6.1.2 Onboard computer

Single board computer (SBC) which are used to control the drone in the flying arena. When a PC is used to control a drone, this referred as **OFFBOARD** control.

A companion computer is referred to SBC that is connected to a flight controller. Usually, SBC is used to perform more sophisticated/high computations that the flight controller can not. In other words, the flight controller is designed for low-level tasks, e.g. attitude control, motor driving, sensor data acquisition. However, the companion computer is used for high-level-control e.g. path planning, optimization.

6.2 Motion Capture Setup: OptiTrack

6.2.1 Camera calibration (skip for bootcamp)

Make sure that you remove any markers from the captured area and Area-C before performing calibration.

Make sure that you use clean markers on the Wanding stick.

The calibration involves three main steps

- Sample collections using the Wanding stick
- Ground setting using the L-shape tool
- Ground refinement

Follow this guide in order to perform the calibration.

Note: It is recommended to perform camera calibration on a weekly basis, or every couple of weeks.

6.3 OptiTrack Interface to ROS

Getting positions of objects in the observable OptiTrack space to ROS works as follows.

6.3.1 Required Hardware

- Mocap machine. Runs Motive Motion Capture Software.
- Optitrack Motion Capture System
- WiFi router (5GHz recommended)
- A Linux based computer, normal PC or on-board embedded computer like ODROID XU4 will work. The Linux computer should be connected to the router either via Ethernet cable or WiFi connection.

6.3.2 Required Software

- Motive. It allows you to calibrate your OptiTrack system, stream tracking information to external entities.
- ROS Kinetic installed on your Linux computer.
- The package vrpn_client_ros for ROS to receive the tracking data from the Mocap computer.

6.3.3 Installation

Odroid XU4

Download Ubuntu 16 with ROS Kinetic minimal.

Flash image with Etcher to ODROID XU4 eMMC.

Important: Make sure that you expand your eMMC card after you flash a new image in order to use the full space of the eMMC card. Use Gparted Partition Editor on Linux to merge unallocated space with flashed space. Choose your eMMC from the dropdown list on the right, select your partition and click `Resize/Move`. Click on the right black arrow and drag it until the partition has its new (desired) size, then click on the `Resize/Move` button. Click apply and wait until it will resize the partition.

Now connect your ODROID XU4 to monitor using HDMI cable. You will also need a keyboard.

After powering the ODROID you will prompt to enter username and password. It's all `odroid`. Plug the WiFi Module 4 to the ODROID's USB port.

Check the WiFi card number by typing following command

```
ifconfig -a
```

To set a static IP address open `/etc/network/interfaces` file for editing by following command

```
sudo nano /etc/network/interfaces
```

Add or edit following lines to the file, and make sure it matches your WiFi network. Added lines should look similar to this.

```
auto wlan0 # The following will auto-start connection after boot
allow-hotplug wlan0 # wlan0 WiFi card number
iface wlan0 inet static
address 192.168.0.xxx # Choose a static IP, usually you change the last number only␣
↪for different devices
netmask 255.255.255.0
```

(continues on next page)

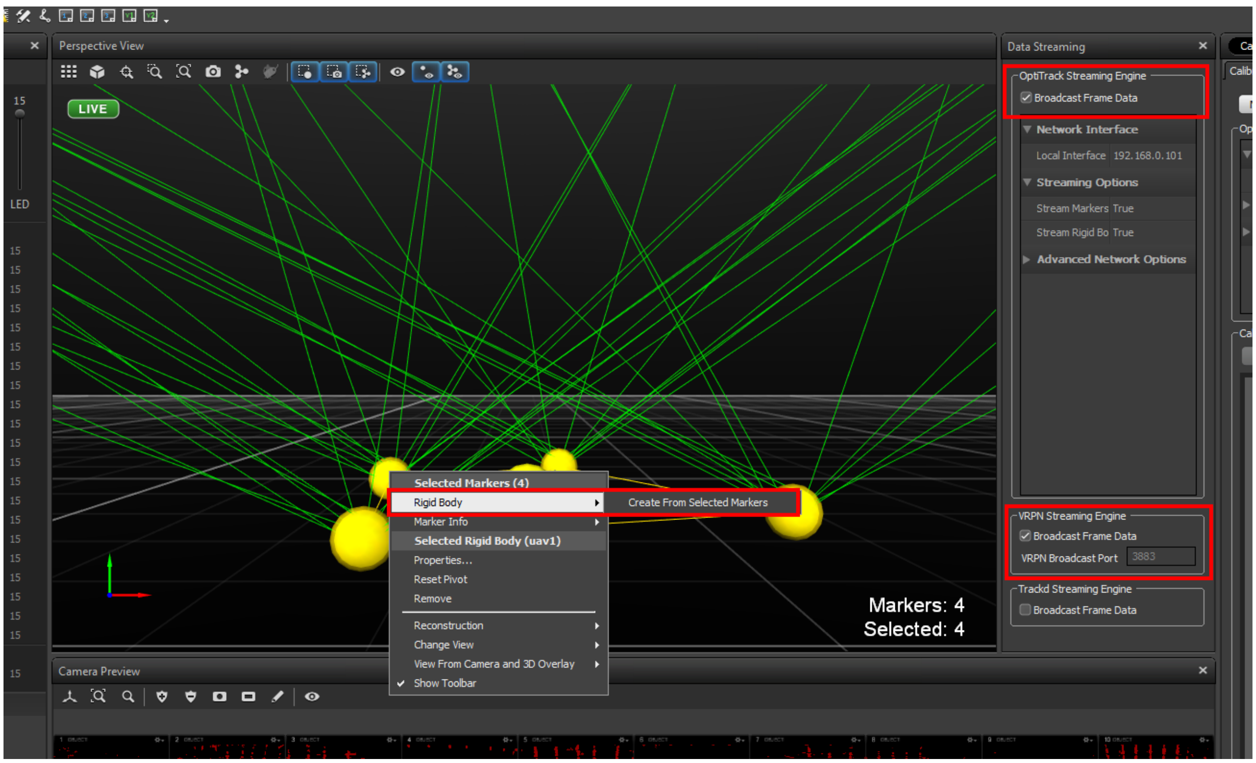
(continued from previous page)

```
broadcast 192.168.0.255
gateway 192.168.0.1 # Your router IP address
dns-nameservers 8.8.8.8
wpa-ssid "RISC-AreaC" # WiFi name (case sensitive)
wpa-psk "risc3720" # WiFi password
```

Mocap computer settings

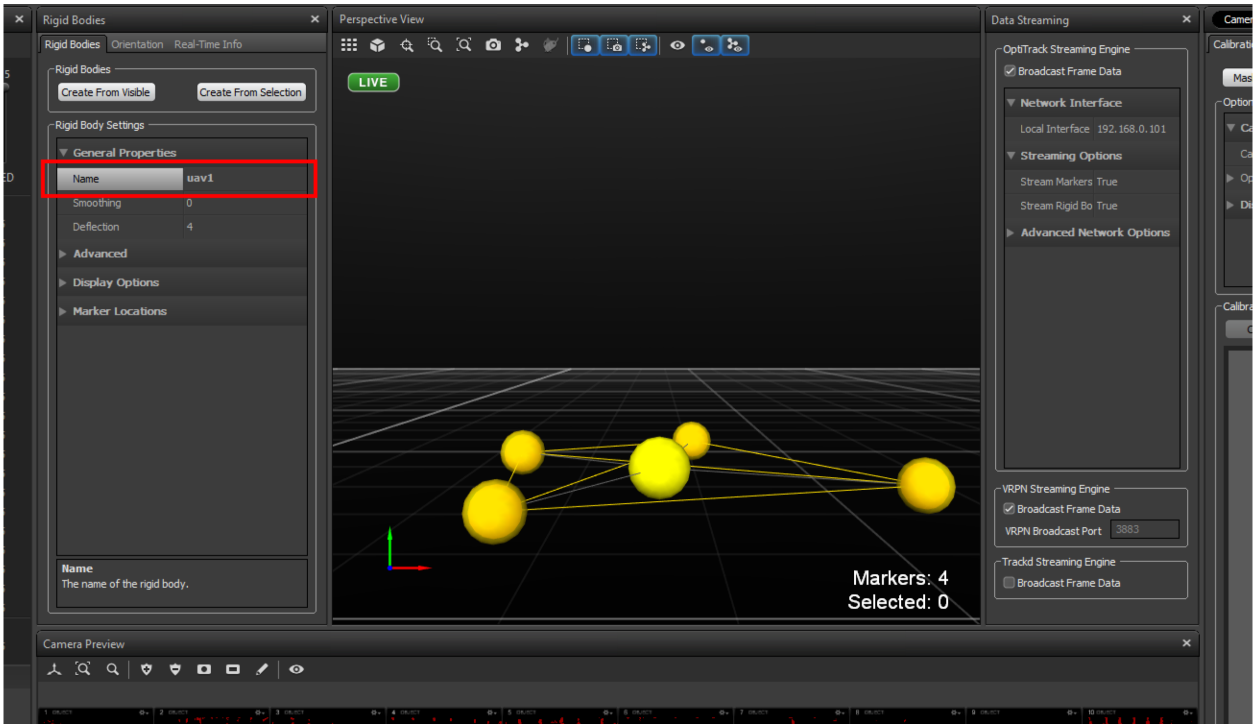
In Motive, choose **View > Data Streaming** from menu bar. Check the boxes `Broadcast Frame Data` in **OptiTrack Streaming Engine** and **VRPN Streaming Engine** sections. Create a rigid body by selecting markers of interest. In **Advanced Network Options** section change `Up Axis` to `Z Up`.

Important: Align your robot's forward direction with the the system +x-axis.



Make sure you either turn off the Windows Firewall or create outbound rules for the VRPN port (recommended).

Right click on the body created, choose **Properties** and rename it such that there is no spaces in the name.



6.3.4 Streaming MOCAP Data

Check the IP address assigned to the Mocap machine, in our case it's **192.168.0.101**

On your odroid), where you want to get tracking data, run the `vrpn_client_ros` node as follows

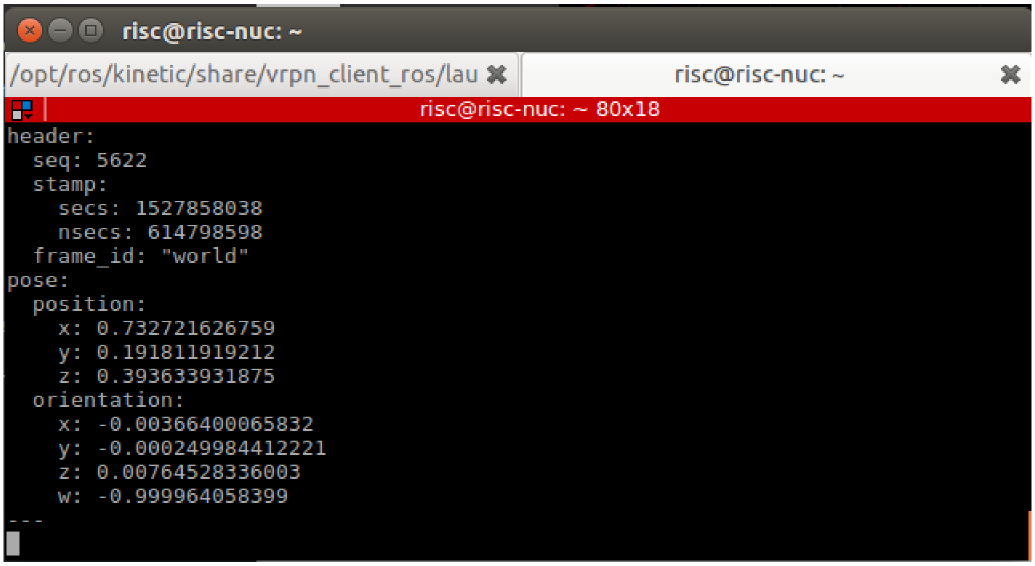
```
roslaunch vrpn_client_ros sample.launch server:=192.168.0.101
```

Now you should be able to receive Mocap data under topic `/vrpn_client_node/<rigid_body_name>/pose`.

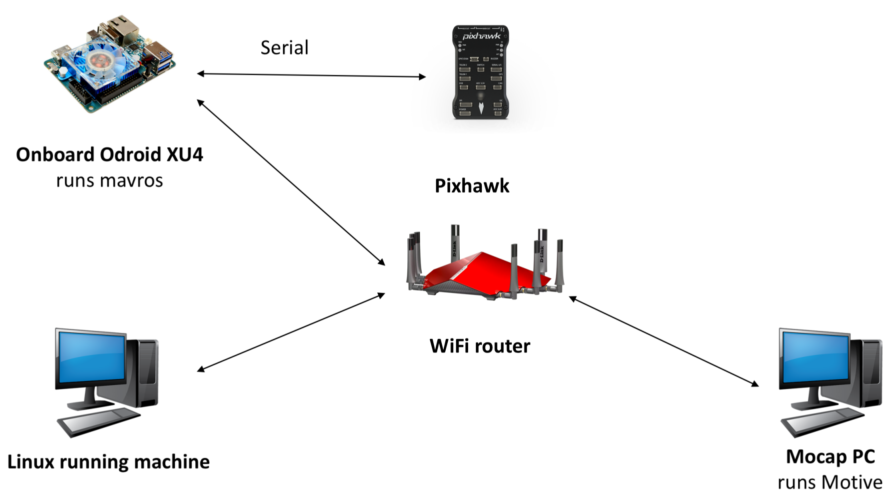
Open new terminal (**CTRL + ALT + F2/F3/F3…**) and try following command

```
rostopic echo vrpn_client_node/<rigid_body_name>/pose
```

You should get similar to this. More information on message type here.



6.4 Feeding MOCAP data to Pixhawk



6.4.1 Intro

This tutorial shows you how to feed MOCAP data to Pixhawk that is connected to an ODROID, or an on-board linux computer. This will allow Pixhawk to have indoor position and heading information for position stabilization.

6.4.2 Hardware Requirements

- Pixhawk or similar controller that runs PX4 firmware
- ODROID (we will assume XU4)
- Serial connection, to connect ODROID to Pixhawk. You will need to solder you own USB/FTDI cable to connect from Odroid USB port to `TELEM2` port on Pixhawk. Mind that `TX` connects to `RX`, `RX` connects to

TX, G to G. If you are using **MindPX** flight controller, just use a USB to micro-USB cable and connect it to **USB/OBC** port.

- OptiTrack PC
- WiFi router (5GHz is recommended)

6.4.3 Software Requirements

- Linux Ubuntu 16 installed on ODROID XU4. A minimal image is recommended for faster executions.
- ROS Kinetic installed on ODROID XU4. Already preinstalled in the image.
- MAVROS package: Binary installation. Already preinstalled in the image.
- Install vrpn_client_ros package. Already preinstalled in the image.

Now, you need to set your flight controller firmware PX4, to accept mocap data. EKF2 estimator can accept mocap data as vision-based data.

6.4.4 Settings in QGroundControl

To set up the default companion computer message stream on TELEM 2, set the following parameters:

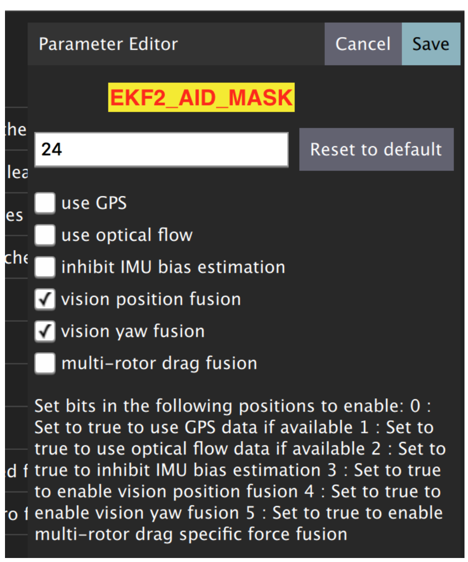
If using firmware version below 1.9.0, change the following parameters:

- SYS_COMPANION = Companion Link (921600 baud, 8N1)

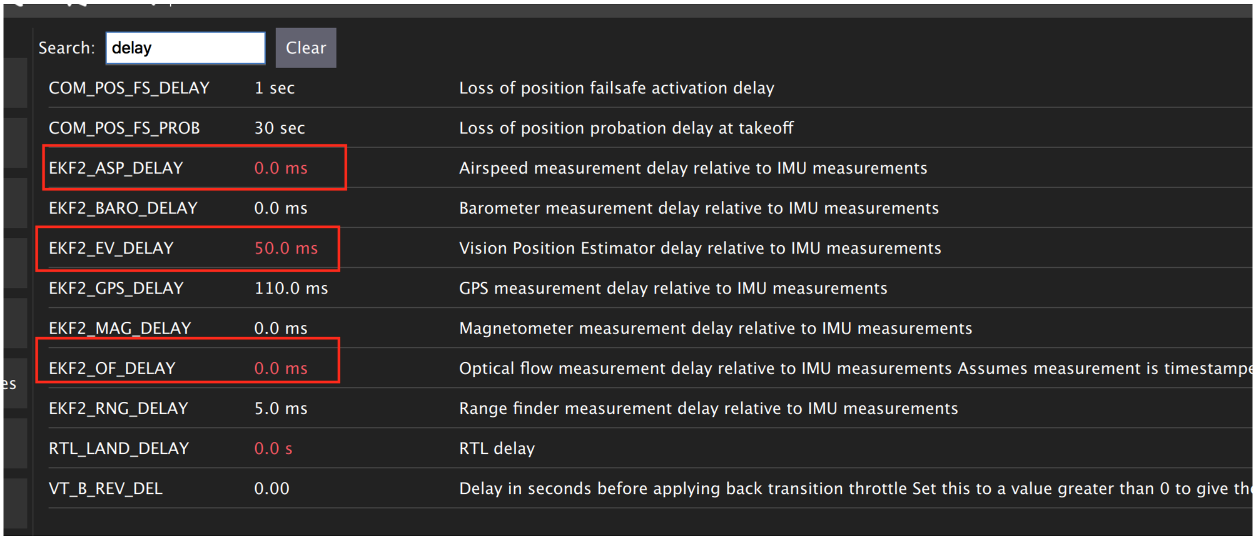
Starting from firmware 1.9.0, change the following parameters:

- MAV_1_CONFIG = TELEM 2 (MAV_1_CONFIG is often used to map the TELEM 2 port)
- MAV_1_MODE = Onboard
- SER_TEL2_BAUD = 921600 (921600 or higher recommended for applications like log streaming or FastRTPS)

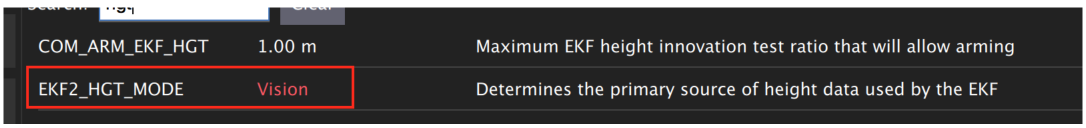
Set EKF2_AID_MASK to **not** use GPS, and use **vision position fusion** and **vision yaw fusion**.



There are some delay parameters that need to set properly, because they directly affect the EKF estimation. For more information read this wiki



Choose the height mode to be vision



(OPTIONAL, for better accuracy). Set the position of the center of the markers (that define the rigid body in the mocap system) with respect to the center of the flight controller. +x points forward, +y right, +z down

Now Restart Pixhawk

6.4.5 Getting MOCAP data into PX4

Assuming your `vrpn_client_node` is still running from optitrack-interface on your ODROID, we will republish it to another topic by `relay` command.

You will need to run MAVROS node by openning a new separate terminal on ODROID (CTRL + ALT + F2/F3/F4)

```
roslaunch mavros px4.launch fcu_url:=/dev/ttyUSB0:921600 gcs_url:=udp://@192.168.0.
↪105:14550
```

where `fcu_url` is the serial port that connects ODROID to the flight controller. Use `ls /dev/ttyUSB*` command on your Odroid to see if serial port is connected. Parameters `gcs_url:=udp://@192.168.0.119:14550` is used to allow you to receive data to `QGroundControl` on your machine (that has to be connected to the same WiFi router). Adjust the IP to match your PC IP, that runs `QGroundControl`.

MAVROS provides a plugin to relay pose data published on `/mavros/vision_pose/pose` to PX4. Assuming that MAVROS is running, you just need to remap the pose topic that you get from Mocap `/vrpn_client_node/<rigid_body_name>/pose` directly to `/mavros/vision_pose/pose`.

```
rosrun topic_tools relay /vrpn_client_node/<rigid_body_name>/pose /mavros/vision_pose/
↪pose
```

Check whether if you can switch your drone to **Position** mode (will be reported in `QGroundControl`). If successfull, you are ready to use position hold/offboard modes.

Check this page before first flight in Position mode.

6.4.6 Checking EKF2 Consistency via Log Files (optional)

Please refer to this link

6.5 Flying

6.5.1 Intro

Now it's time to fly your drone in the cage!

We will need a PC running Linux with Joystick connected to it. To establish ODROID communication with that PC, we will setup ROS Network. The Odroid on the drone will be the ROS Master. The logic is the same as in the Software in the Loop simulator. The joystick commands will be converted to position setpoints and will be published to `/mavros/setpoint_raw/local` node. Finally MAVROS will send setpoints to autopilot (real flight controller on your drone).

6.5.2 Setup a ROS Network

- First let's tell NUC/laptop running Linux that Odroid is the **Master** in the ROS network by editing `.bashrc` file. Open terminal and open `.bashrc` file for editing.

```
gedit ~/.bashrc
```

- Add following lines to the end of the file. Just change last numbers to corresponding IP numbers.

```
export ROS_MASTER_URI=http://192.168.0.odroid_ip_number:11311
export ROS_HOSTNAME=192.168.0.pc_ip_number
```

Make sure you **source** the `.bashrc` file after this.

- From NUC/laptop log into an ODROID to get access to a command-line over a network. We will setup an Odroid as a Master now.

```
ssh odroid@192.168.0.odroid_ip_number
```

It will prompt to enter password, if you use minimal image provided then it's **odroid**.

- Let's edit `.bashrc` file on ODROID as well.

```
nano .bashrc
```

- Add the following lines to the end of the file. Just change last numbers to corresponding IP numbers.

```
export ROS_MASTER_URI=http://192.168.0.odroid_ip_number:11311
export ROS_HOSTNAME=192.168.0.odroid_ip_number
```

To save file, press Ctrl+X, press Y, hit Enter. Source the `.bashrc` file.

6.5.3 ODROID commands

- Run on ODROID separate terminals `vrpn_client_ros`, `MAVROS` and relay.

```
roslaunch vrpn_client_ros sample.launch server:=192.168.0.101
```

```
roslaunch mavros px4.launch fcu_url:=/dev/ttyUSB0:921600 gcs_url:=udp://@192.168.0.pc_
↪ip_number:14550
```

```
rosrun topic_tools relay /vrpn_client_node/<rigid_body_name>/pose /mavros/vision_pose/
↪pose
```

6.5.4 NUC/laptop commands

It's important at this stage to check if data from Mocap is published to `/mavros/vision_pose/pose` and `/mavros/local_position/pose` by echo'ing these topics.

- Download `joystick_flight.launch` and `setpoints_node.py` files to the NUC/laptop and put them into `scripts` and `launch` folder accordingly. Find and understand what's different from code in SITL files.

```
# Inside the scripts folder of your package
wget https://raw.githubusercontent.com/risckaust/risc-documentations/master/src/
↪indoor-flight/setpoints_node.py

#Inside the launch folder of your package
wget https://raw.githubusercontent.com/risckaust/risc-documentations/master/src/
↪indoor-flight/joystick_flight.launch
```

- Make sure you give permissions to the joystick.

Danger: Keep the transmitter nearby to engage the `Kill Switch` trigger in case something will go wrong.

- Now run in a new terminal on the NUC/laptop your launch file

```
roslaunch mypackage joystick_flight.launch
```

6.5.5 Joystick control

`BUTTON 1` - Arms the quadcopter

`BUTTON 3` - Switches quadcopter to OFFBOARD flight mode. It should takeoff after this.

`BUTTON 2` - Lands the quadcopter

`BUTTON 11` - Disarms the quadcopter

Enjoy your flight.

Mohamed Abdelkader and Kuat Telegenov.

CHAPTER 7

The software in the Loop Rover Control

This tutorial explains the steps required to drive a simulated rover in the Gazebo simulator. We are going to learn an essential way of controlling the rover by publishing the desired setpoints to a specific topic. There is a mode in PX4 autopilot, which is called **OFFBOARD** mode. This mode allows the autopilot to accept specific external commands such as position, velocity, or attitude setpoints.

In general, a MAVROS node provides setpoint plugins which will listen to user input on specific setpoint topics. Once the user publishes to those specific setpoint topics and if the mode set to **OFFBOARD**, the MAVROS node will transfer those setpoints to the autopilot to execute.

In this tutorial, we will send position setpoints to the autopilot via a setpoint topic that is available in MAVROS. Once set points are received on that topic, the MAVROS node will send it to the autopilot. The setpoint topic that we will use in this tutorial is `mavros/setpoint_position/local`. Next, we will create our custom simple ROS package in which we create a simple ROS node that will publish setpoints one after one to follow the square. Finally, MAVROS will take the position setpoints and send them to the autopilot to execute.

7.1 Hardware Requirements

- Desktop Linux Machine with a minimum of 8GB RAM, 16GB recommended, Ubuntu 16.04 installed

7.2 Software Requirements

- **Ubuntu 16.04**
- **ROS Kinetic** (full desktop installation)
- **Gazebo 7**: will be automatically installed with ROS
- **PX4 firmware** installation on Linux: Autopilot software which includes the software-in-the-loop firmware
- **MAVROS** package: Autopilot ROS interface

Note: In this tutorial, it is assumed that the reader is familiar with basic Linux commands, ROS Basics.

7.3 Setup Steps

- Install `QGroundControl` from here. Use the AppImage option.

7.4 SITL with Gazebo

There are launch files available to run the simulation wrapped in the ROS.

To run SITL wrapped in ROS with Rover configuration, the ROS environment needs to be updated:

```
cd ~/src/Firmware
DONT_RUN=1 make px4_sitl gazebo_rover
cd ~/catkin_ws
catkin build
```

Now close the terminal.

7.5 Launching Gazebo with ROS Wrappers

Now, you are ready to launch Gazebo + PX4 SITL app + ROS + MAVROS. To do that, execute the following command.

```
roslaunch px4 mavros_posix_sitl.launch
```

If everything launched correctly, you should see a drone in the simulated environment. To change vehicle to the rover, relaunch the previous command with the specified argument for a vehicle. By default, value is set as *iris*.

```
roslaunch px4 mavros_posix_sitl.launch vehicle:="rover"
```

You should be able to see many `/mavros/...` topics using `rostopic list` in a new terminal. Also if you execute `rosnode list` in a new terminal, you should see the following.

```
$ rosnode list
/gazebo
/gazebo_gui
/mavros
/rosout
```

To double-check that MAVROS node is connected correctly to the PX4 SITL app, try to `echo` some topics _e.g._

```
rostopic echo /mavros/state
```

This will show if the MAVROS node is connected to the PX4 SITL or not.

Now, you can monitor the rover's states and control it via a MAVROS node.

7.6 Custom Setpoint Node

Now, it's time for some coding! You will write a ROS node in Python that publishes the desired position setpoints into `mavros/setpoint_position/local`.

Publishing to `mavros/setpoint_position/local` topic is not enough to get the autopilot to track the setpoints. It has to be in **OFFBOARD** mode. So, in your custom node, you will have to send a signal to activate this mode, only once. You need to **remember** that for this mode to work, you will need to be publishing setpoints beforehand, then, activate it, and continue publishing setpoints. **If you don't publish setpoints at more than 2Hz, it will go into a failsafe mode** and **OFFBOARD** mode will be off.

First, create your custom ROS package. The code is commented so you can get an idea of what each part does. Go through code and try to understand it!

```
cd ~/catkin_ws/src
catkin_create_pkg mypackage std_msgs mavros_msgs roscpp rospy
cd mypackage
# usually python scripts (nodes) are placed in a folder called scripts
mkdir scripts
cd scripts
wget https://raw.githubusercontent.com/risckaust/risc-documentations/master/src/
↪gazebo-rover/square.py
```

Make the python file an executable,

```
chmod +x square.py
```

The python file is missing the topic name for the publisher. Your goal is to fix it by providing the right name for the topic.

Make a **launch** folder. We will create a ROS launch file to run everything at once. Open the launch file and understand what every line executes.

```
cd ~/catkin_ws/src/mypackage
mkdir launch
cd launch
wget https://raw.githubusercontent.com/risckaust/risc-documentations/master/src/
↪gazebo-rover/main.launch
```

This launch includes MAVROS sitl launch file. But you still need to change parameter for the vehicle, so it spawns the rover into the simulated world.

Build and source the catkin workspace. In a new terminal, you can run the launch file by executing:

```
roslaunch mypackage main.launch
```

Now, you should see a rover following the square autonomously.

7.7 Contributors

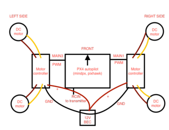
Sarah Toonsi and Kuat Telegenov.

CHAPTER 8

Rover Assembly

8.1 Basic principles

This tutorial will guide you how to build a skid-steer rover. Skid-steer vehicles have the wheels mechanically locked in synchronization on each side, and where the left-side drive wheels can be driven independently of the right-side drive wheels. We have to set skid-steer frame in flight controler's configuration. The flight controller will generate PWM signal at MAIN PWM output ports. We will connect these ports to motor controller input (RC1 in this case). Then motor controller will drive the DC motors according to the signal coming from the flight controller. The following diagram shows how the system components connected together.



While building the rover, feel free to place the components anywhere inside the frame but take care of wires. Carefully choose zipties, shrinking tubes, double sided tapes or soldering for different situations.

8.2 Preliminaries

This tutorial assumes you have the following skills:

- *ROS Basics*.
- Soldering, if not, please refer to basic skill video.
- Basic knowledge about LiPo batteries. Answer the following questions. You may read this article.
  - What do 3s, 4s mean?
  - What does 20c mean?
  - What does 1400mAh mean?
  - What are the parameters of your battery?

- How to charge LiPo battery? How to measure it voltage using battery meter?
- What's the minimum voltage to use a LiPo on the robot?

Danger: Do not leave your battery plugged in your robot for a long time and never discharge a LiPo battery below 3.4V per cell.

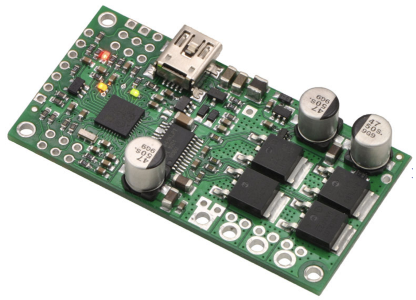
8.3 Hardware assembly

8.3.1 Introduction

You will need

- Rover frame with four wheels and DC motors.

- Two **motor controllers** to control DC motors.



- Flight Controller. Use any flight controller available in the lab. Just make sure you have compatible power modules, receivers, GPS, and other additional modules. The documentations for each board are available here.
- Remote control system. A remote control (RC) radio system is required if you want to manually control your vehicle. In addition to the transmitter/receiver pairs being compatible, the receiver must also be compatible with PX4 and the flight controller hardware. It's recommended to use **Taranis X9D Plus transmitter with X8R receiver** as shown below

- UBEC (Universal Battery eliminator circuit) to convert voltage to power Odroid. A BEC is basically a step down voltage regulator. It will take your main battery voltage (e.g. 11.1 Volts) and reduce it down to 5/12 Volts to safely power your Odroid and other electronics. We will use Twin Output BEC which will power Odroid and Motor controllers at the same time.

- Power module. It is the best way to provide power for flight controller unit. It has voltage and current sensors that allows autopilot to estimate remaining battery charge precisely. Usually it comes with every autopilot controller as a default kit. Check official documentations to match right power module to a selected flight controller.

- LiPo battery. 3000/4000 mAh 3S battery is recommended.

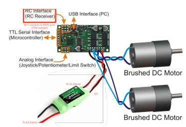
- Odroid XU4. Onboard computer that will run high level programs and algorithms. It will be connected to Flight Controller through serial connection. Odroid will need WiFi USB module, eMMC memory module and DC Plug Cable.

8.3.2 Assembly process

- Install DC motors to the frame if necessary. Attach the wheels to the motor shafts with provided screw sets.

- Attach flight controller on the frame. Take a look at your flight controller and make sure the arrow is pointing to the front. To mount the controller to the frame, use thick double side tape to damp the vibrations.
- Connect right side motor's red cable to **OUTB**, and yellow cable to to the **OUTA** of the first motor controller. The motor controller's **RC1** port should be connected to **MAIN1** PWM ouput channel. Make sure you match **SIGNAL**, **+** and **-**. Use servo cable for this connection.
- Connect left side motor's red cable to **OUTA**, and yellow cable to to the **OUTB** of the second motor controller. The motor controller's **RC1** port should be connected to **MAIN3** PWM ouput channel. Again match **SIGNAL**, **+** and **-**.
- Connect BEC's 12V positive and ground outputs to **VIN** and **GND** respectively. You have to connect both motor controllers. They will be powered from the same BEC.

The following diagram shows the connection for one of the sides. **RC Interface (RC Receiver)** is used for connecting to flight controller. **USB Interface** is used for modifying settings on the motor controller and flashing firmware.



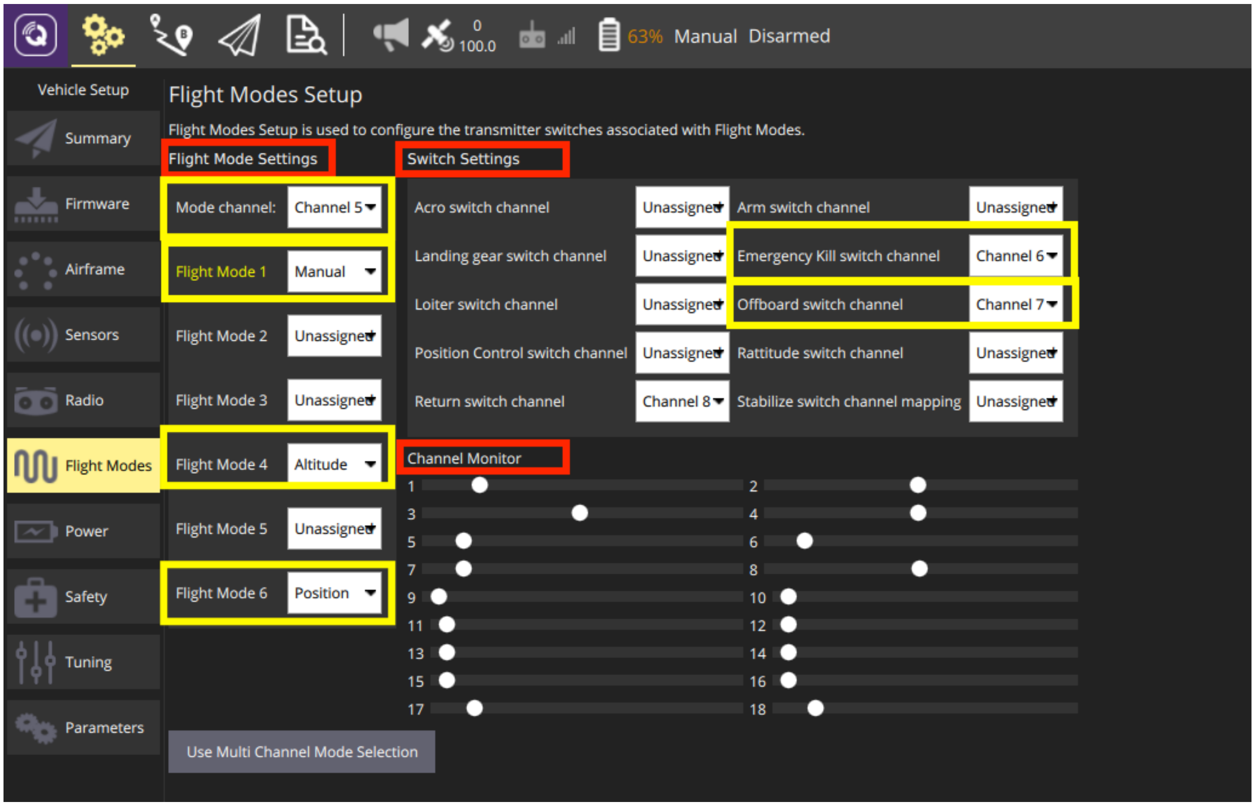
- Install power module on the frame. Plug cable from power module to `POWER` port of your flight controller.
- Plug buzzer and switch to their corresponding ports on flight controller.
- Connect the `RCIN` port from Pixhawk to `SBUS` port on **X8R** and follow the binding process for FrSky X8R.
  - Turn on the **X8R** while holding the **F/S** button on the module. Release the button.
  - Press the **Menu** button on your Taranis X9D transmitter.
  - Go to page 2 by pressing **Page** button.
  - Scroll down with **-** button until you see **Internal RF** line.
  - Select **[Bind]** line, and press **ENT** button. The RED LED on the X8R receiver will flash, indicating the binding process is completed

8.4 Calibration process

- Open **QGroundControl** and connect your flight controller to the computer.
- Install Stable PX4 firmware.
- Set the airframe to Axial Racing AX10. Follow steps from this page.
- Calibrate Compass, Accelerometer, and Level Horizon.
- Calibrate the Radio.
- In `Flight Modes` tab under the **Flight Mode Settings** and **Switch settings** sections set:
  - **Mode Channel** to SB (SB switch labeled on your Taranis X9D)
  - **Mode 1: Manual**.
  - **Mode 4: Altitude**. Climb and drop are controlled to have a maximum rate.
  - **Mode 6: Position**. When sticks are released the vehicle will stop and hold position.
  - **Emergency Kill switch channel** to SF (SF switch labeled on your Taranis X9D). Immediately stops all motor outputs. The vehicle will crash, which may in some circumstances be more desirable than allowing it to continue flying.

   - **Offboard switch channel** to SA (SA switch labeled on your Taranis X9D).

You should have similar as shown in the picture below. Channels for **Flight Mode Settings** and **Switch Settings** might differ.



Hint: If you set everything right, you will see changes in **Flight Mode Settings** section highlighted as yellow. Also, moving sticks, dials and switches will be reported in **Channel Monitor** section.

- In `Power tab` write the parameters of your battery (Number of cells), calibrate the battery voltage.
   - Press **Calculate** on the **Voltage divider** line.
   - Measure the voltage with Digital Battery Capacity Checker by connecting it to the battery.
   - Enter the the voltage value from the Digital Battery Capacity Checker and press **Calculate** button.
- Search for `FW_ARSP_MODE` in QGroundControl parameters, and set it to **Airspeed disabled**.
- Search for `PWM_MAX` and `PWM_MIN` and set them to **2200** and **800** respectively.

8.5 Configuring the motor controller

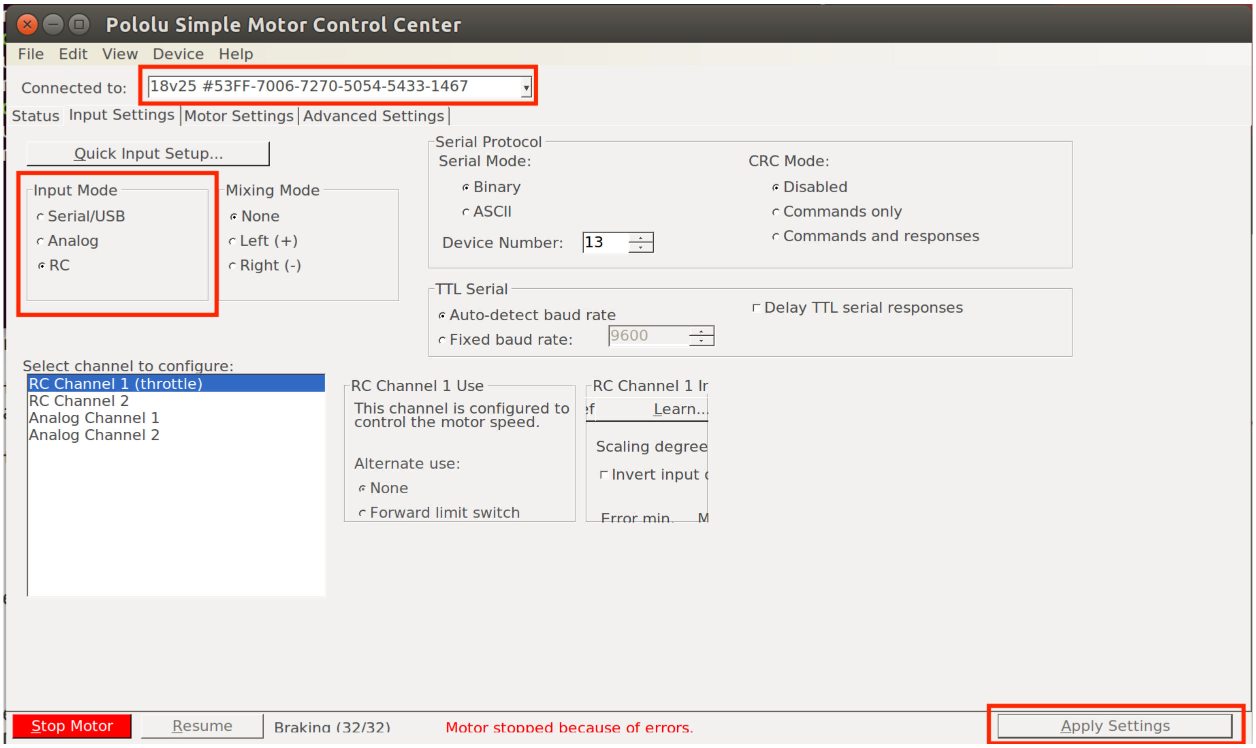
Download Simple Motor Controller Linux Software on the Ubuntu based computer. Open terminal and navigate to the downloaded folder, and unzip the archive with the following command.

```
tar -xzvf smc-linux-101119.tar.gz #File name might differ
```

After following the instructions in **README.txt** , you can run the program by following command.

```
./SmcCenter
```

Connect motor controller to the Ubuntu based computer using mini USB cable. Navigate to **Input Settings** tab and change **Input Mode** to **RC** as shown below. After that press **Apply Settings**.



8.6 Driving the rover with the transmitter

- Make sure you switch **Kill switch** to off. Select **Manual** as your flight mode.
- Check the battery level, make sure it's enough to perform your first ride.

Important: Always check the battery before starting

- Put the rover in the cage.
- Left stick on the transmitter controls left wheels, while right stick controls left wheels. By moving two sticks in the same side (left or ride) will move rover either front or backward. If you move stick in a opposite direction from each other, that will make rover to turn around it's own axis.

8.7 Markers installation

Attach four Motion Capture Markers to the Rigid Body Marker Base. Mount Rigid Body Marker Base on the rover.

8.8 Contributors

Mohammad Albeaik and Kuat Telegenov.

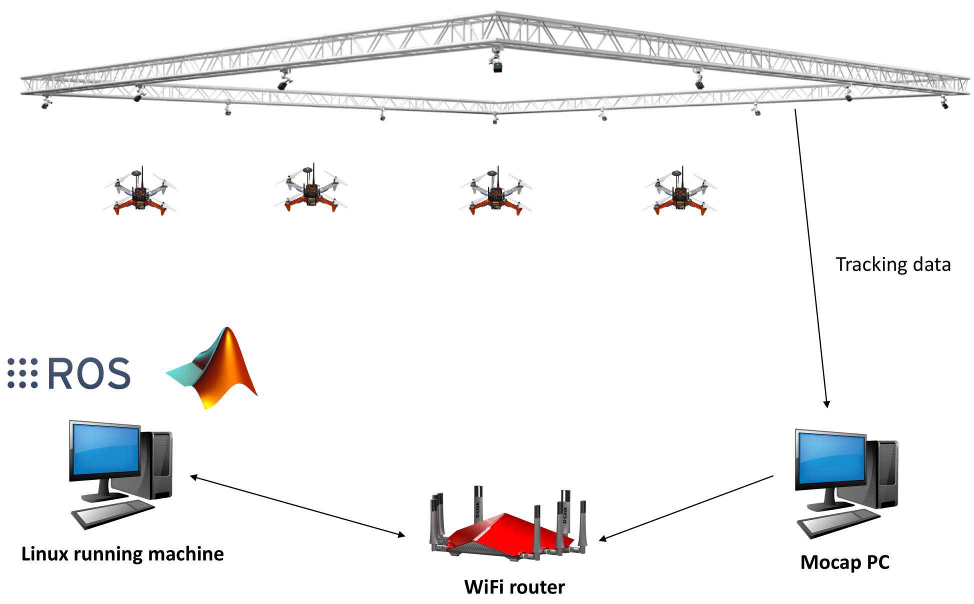
CHAPTER 9

Indoor Rover Navigation

9.1 System Architecture

In order to start navigation rover autonomously, we need the position and orientation of the rover.

This section will guide you how to use OptiTrack Motion Capture System, how to stream position and orientation data to ROS, and feed it to your flight controller. Finally you will be able to navigate rover autonomously inside the arena.



The overall systems has following main elements:

- OptiTrack Motion Capture System
- Object to be tracker (quadcopter or rover) inside the arena
- Controller - onboard computer, usually Odroid

Let's discuss each element in details

9.1.1 Motion capture system

OptiTrack motion capture system (Mocap hereinafter) works as follows. The overhead cameras send out pulsed infrared light using the attached infrared LEDs, which will then be reflected by markers on the object and detected by the OptiTrack cameras. Knowing the position of those markers in perspective of several cameras, the actual 3D position of the markers in the room can be calculated using triangulation. Simply Mocap provides high precision indoor local position and orientation estimation. Position is meters and orientation is in quaternion, which can be converted Euler angles in radians. In RISC lab we use twenty **Prime17w** cameras that are installed in the flying arena.

All cameras are connected to a single Mocap PC through network switches. Motive Optical motion capture software is installed on this PC.

9.1.2 Onboard computer

Single board computer (SBC) which are used to control the rover in the flying arena. When a PC is used to control a drone, this referred as **OFFBOARD** control.

A companion computer is referred to SBC that is connected to a flight controller. Usually, SBC is used to perform more sophisticated/high computations that the flight controller can not. In other words, the flight controller is designed for low-level tasks, e.g. attitude control, motor driving, sensor data acquisition. However, the companion computer is used for high-level-control e.g. path planning, optimization.

In case of rover, PX4 does not have a controller for **OFFBOARD** control. Instead, we will be using **RC_OVERRIDE** option. We will be overriding the RC signals coming from transmitter. Now, rover will be controlled from the Odroid, which will generate needed RC signals to follow the path, or navigate to the setpoint.

9.2 Motion Capture Setup: OptiTrack

9.2.1 Camera calibration (skip for bootcamp)

Make sure that you remove any markers from the captured area and Area-C before performing calibration.

Make sure that you use clean markers on the Wanding stick.

The calibration involves three main steps

- Sample collections using the Wanding stick
- Ground setting using the L-shape tool
- Ground refinement

Follow this guide in order to perform the calibration.

Note: It is recommended to perform camera calibration on a weekly basis, or every couple of weeks.

9.3 OptiTrack Interface to ROS

Getting positions of objects in the observable OptiTrack space to ROS works as follows.

9.3.1 Required Hardware

- Mocap machine. Runs Motive Motion Capture Software.
- Optitrack Motion Capture System
- WiFi router (5GHz recommended)
- A Linux based computer, normal PC or on-board embedded computer like ODROID XU4 will work. The Linux computer should be connected to the router either via Ethernet cable or WiFi connection.

9.3.2 Required Software

- Motive. It allows you to calibrate your OptiTrack system, stream tracking information to external entities.
- ROS Kinetic installed on your Linux computer.
- The package vrpn_client_ros for ROS to receive the tracking data from the Mocap computer.

9.3.3 Installation

Odroid XU4

Download Ubuntu 16 with ROS Kinetic minimal image on your Ubuntu based computer.

Flash downloaded image with Etcher to ODROID XU4 eMMC. Use proper microSD adapters to plug it to your computer.

Next, expand your eMMC card to use the full space of the eMMC card. Use **Gparted Partition Editor** on Linux to merge unallocated space with flashed space. Choose your eMMC from the dropdown list on the right, select your partition and click `Resize/Move`. Click on the right black arrow and drag it until the partition has its new (desired) size, then click on the `Resize/Move` button. Click apply and wait until it will resize the partition.

Now connect your ODROID XU4 to monitor using HDMI cable. You will also need a keyboard, and 5V/4A power supply.

After powering the ODROID you will prompt to enter username and password. It's all `odroid`. Plug the WiFi Module 4 to the ODROID's USB port.

Check the WiFi card number by typing following command

```
ifconfig -a
```

To set a static IP address open `/etc/network/interfaces` file for editing by following command

```
sudo nano /etc/network/interfaces
```

Add or edit following lines to the file, and make sure it matches your WiFi network. Added lines should look similar to this.

```
auto wlan0 # The following will auto-start connection after boot
allow-hotplug wlan0 # wlan0 WiFi card number
iface wlan0 inet static
address 192.168.0.xxx # Choose a static IP, range for xxx is 10-254
netmask 255.255.255.0
broadcast 192.168.0.255
gateway 192.168.0.1 # Your router IP address
dns-nameservers 8.8.8.8
```

(continues on next page)

(continued from previous page)

```
wpa-ssid "RISC-AreaC" # WiFi name (case sensitive)
wpa-psk "risc3720" # WiFi password
```

Save the file, reboot the Odroid, disconnect/connect WiFi Module, and check if you can ping any computer in the RISC network.

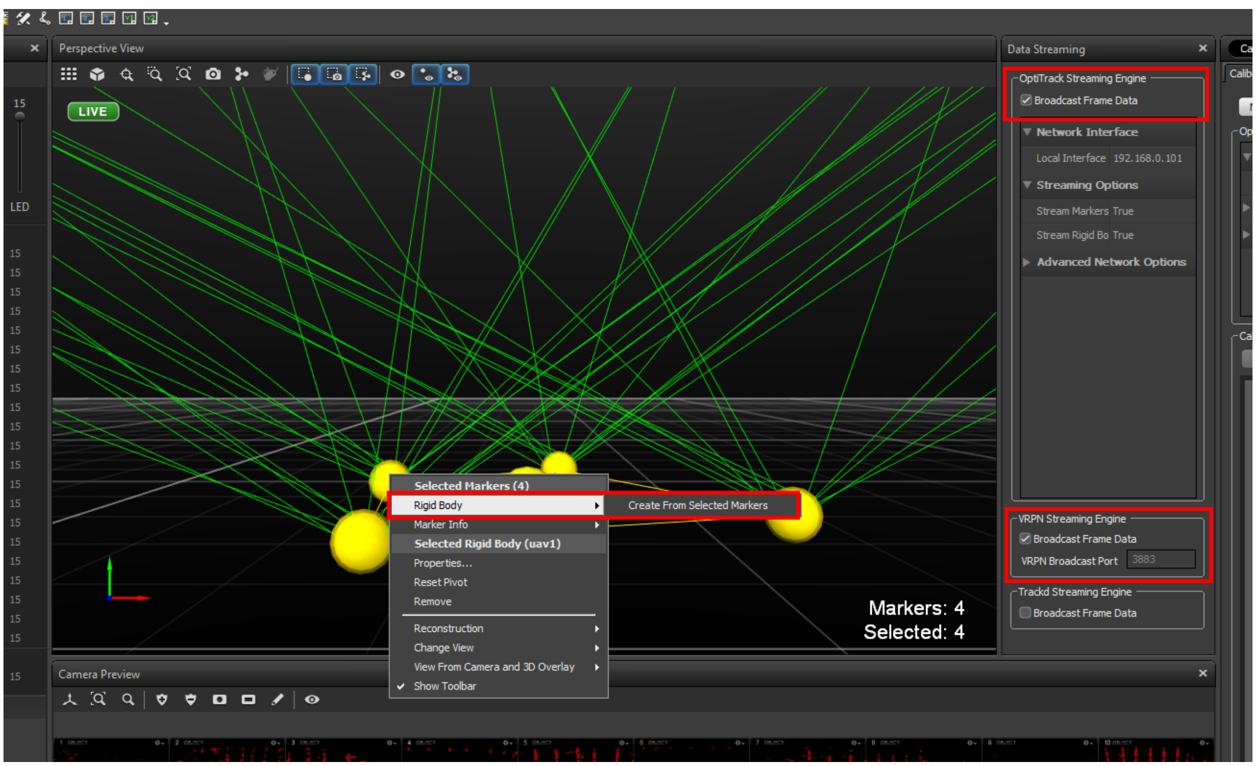
```
ping 192.168.0.101 # Mocap computer
ping 192.168.0.1 # router
```

If you're receiving a reply, the network is set correctly. Power off odroid, disconnect monitor, power and keyboard. From now on we will use **ssh** command to access Odroid' terminal over WiFi.

Mocap computer settings

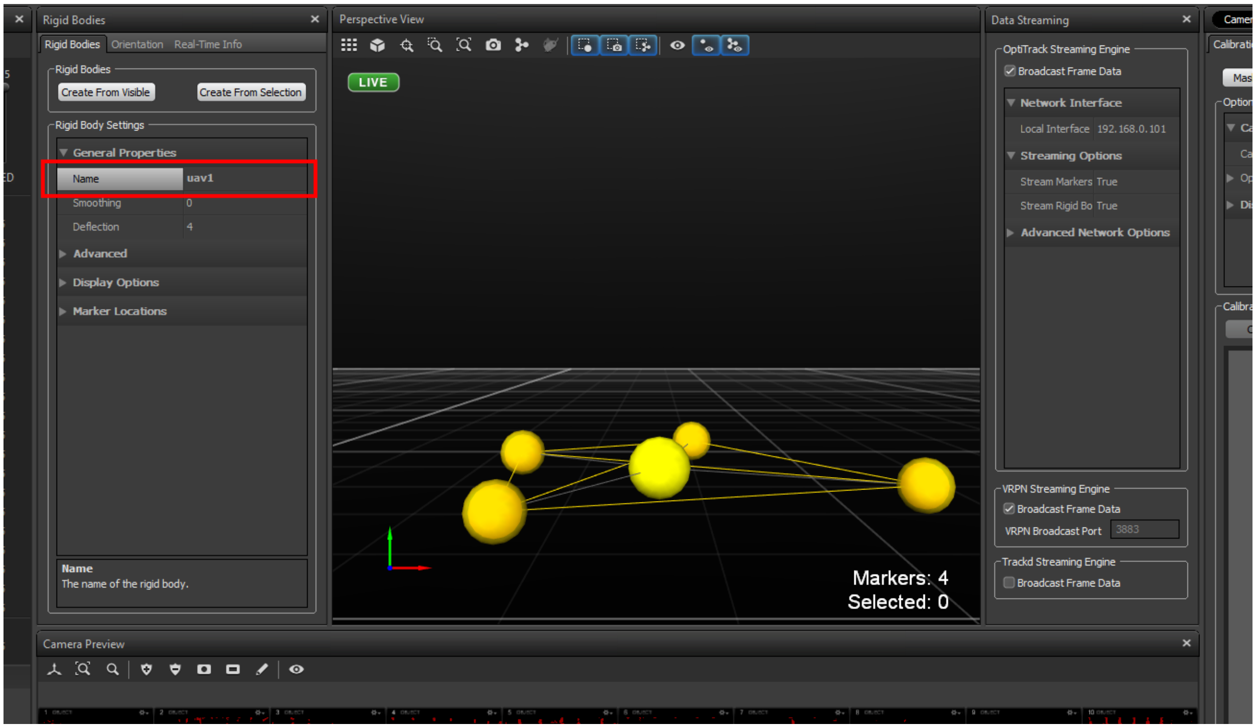
In Motive, choose **View > Data Streaming** from menu bar. Check the boxes `Broadcast Frame Data` in **OptiTrack Streaming Engine** and **VRPN Streaming Engine** sections. Place your Rigid Body Marker Base or the rover (if markers are installed on the rover) inside the cage. Create a rigid body by selecting markers of interest as show in the figure below. In **Advanced Network Options** section change `Up Axis` to `Z Up`.

Important: Align your robot's forward direction with the the system +x-axis.



Make sure you either turn off the Windows Firewall or create outbound rules for the VRPN port (recommended).

Right click on the body created, choose **Properties** and rename it such that there is no spaces in the name.



9.3.4 Streaming MOCAP Data

Check the IP address assigned to the Mocap machine, in our case it's **192.168.0.101**

Power the Odroid, and use Ubuntu based computer in the lab (NUC or laptop). We will remotely access Odroid from another computer connected to the same network. Execute following command from the terminal

```
ssh odroid@192.168.0.xxx # the IP address you set on the Odroid previously
```

It will prompt the password, it's *odroid*.

Now we are in the command-line of the Odroid.

To get the tracking data we need to run the `vrpn_client_ros` node as follows

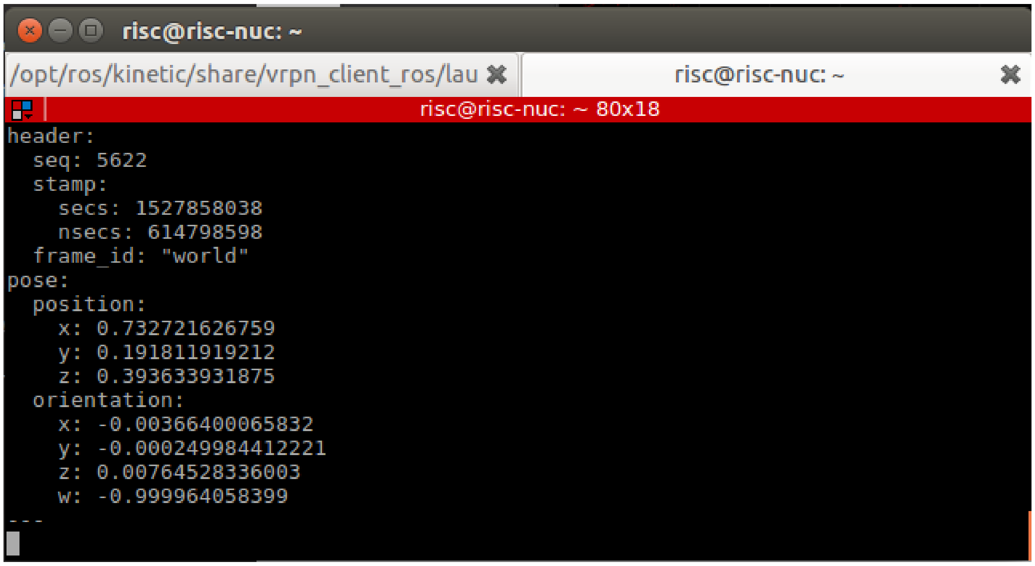
```
roslaunch vrpn_client_ros sample.launch server:=192.168.0.101
```

Now you should be able to receive Mocap data under topic `/vrpn_client_node/<rigid_body_name>/pose`.

Open new terminal, *ssh* again, and try following command.

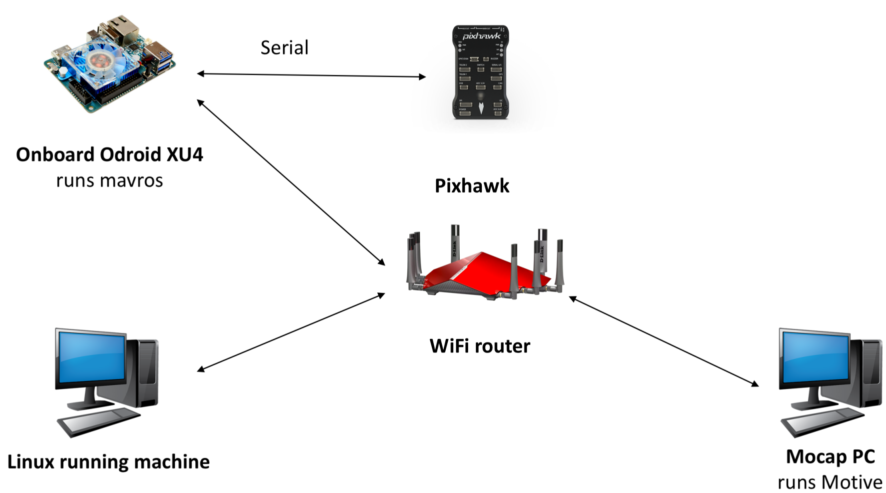
```
rostopic echo vrpn_client_node/<rigid_body_name>/pose
```

You should get similar to this. More information on message type here.



That means you are able to track you rigid body inside the arena, and get the data to the Odroid.

9.4 Feeding MOCAP data to Pixhawk



9.4.1 Intro

This tutorial shows you how to feed MOCAP data to Pixhawk that is connected to an ODROID, or an on-board linux computer. This will allow Pixhawk to have indoor position and heading information for position stabilization.

9.4.2 Hardware Requirements

- Pixhawk or similar controller that runs PX4 firmware
- ODROID (we will assume XU4)

- Serial connection between Odroid and Pixhawk.
- OptiTrack PC
- WiFi router (5GHz is recommended)

9.4.3 Software Requirements

- Linux Ubuntu 16 installed on ODROID XU4. A minimal image is recommended for faster executions.
- ROS Kinetic installed on ODROID XU4. Already preinstalled in the image.
- `MAVROS` package: Binary installation. Already preinstalled in the image.
- Install `vrpn_client_ros` package. Already preinstalled in the image.

Now, you need to set your flight controller firmware PX4, to accept mocap data. `EKF2` estimator can accept mocap data as vision-based data.

9.4.4 Settings in QGroundControl

To set up the default companion computer message stream on `TELEM 2`, set the following parameters:

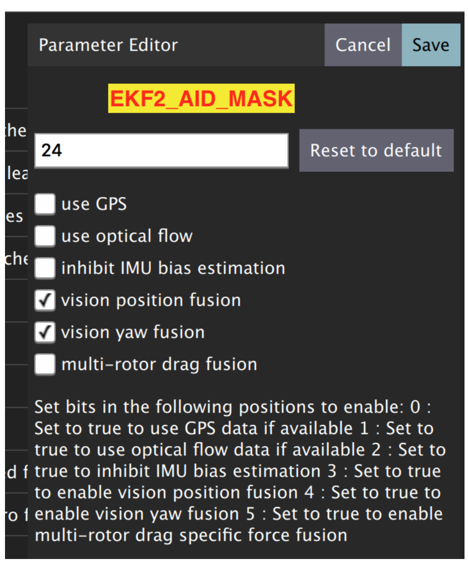
If using firmware version below 1.9.0, change the following parameters:

- `SYS_COMPANION` = Companion Link (921600 baud, 8N1)

Starting from firmware 1.9.0, change the following parameters:

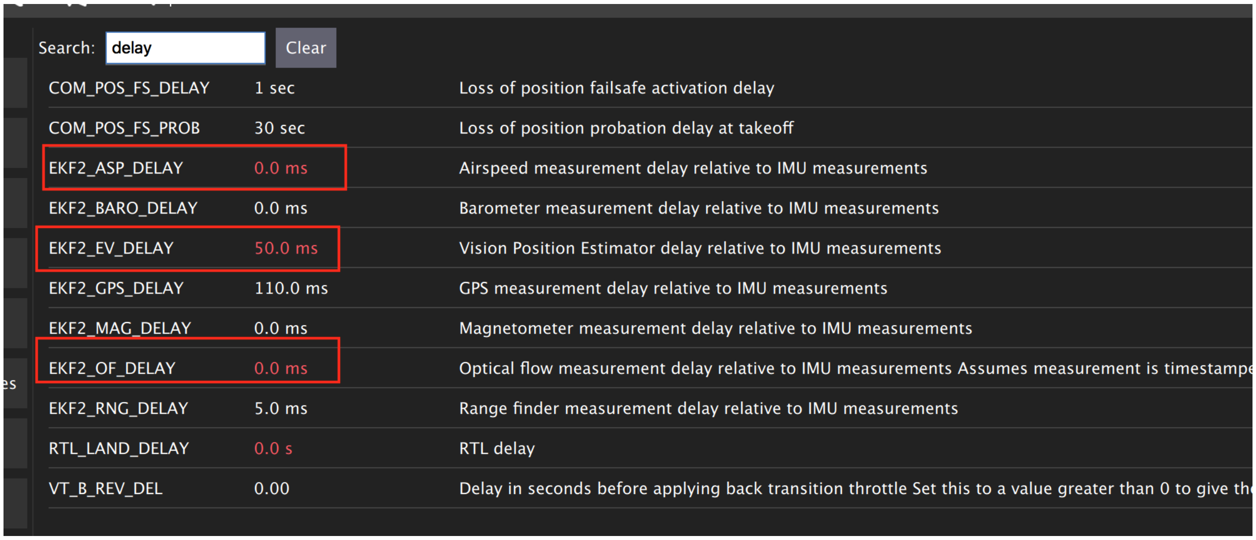
- `MAV_1_CONFIG` = TELEM 2 (MAV_1_CONFIG is often used to map the TELEM 2 port), reboot
- `MAV_1_MODE` = Onboard
- `SER_TEL2_BAUD` = 921600 (921600 or higher recommended for applications like log streaming or FastRTPS)

Set `EKF2_AID_MASK` to **not** use GPS, and use **vision position fusion** and **vision yaw fusion**. This way the rover will use vision for positioning.

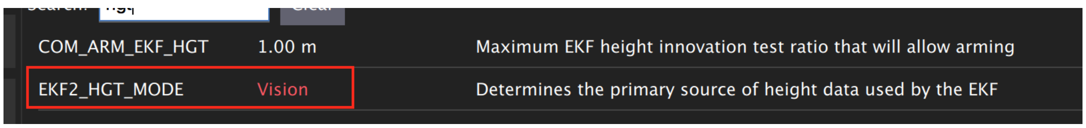


To Enable RC stick override of auto modes, search for `COM_RC_OVERRIDE` in QGroundControl parameters and enable it.

There are some delay parameters that need to set properly, because they directly affect the EKF estimation. For more information read this wiki



Choose the height mode to be vision



(OPTIONAL, for better accuracy). Set the position of the center of the markers (that define the rigid body in the mocap system) with respect to the center of the flight controller. +x points forward, +y right, +z down

Now Restart Pixhawk

9.4.5 Getting MOCAP data into PX4

Odroid installation

It's time to mount Odroid on the rover, and connect it to the Pixhawk.

First, to power the Odroid we need to provide 5V power to it. Solder Odroid DC Plug Cable to female servo cable and connect to the UBEC 5V output cable

Next we need to connect Odroid to the flight controller using serial connection. In case of MindPX simply connect micro-USB cable to `USB/OBC` from the Odroid USB port. In case of Pixhawk use FTDI module. Use servo cable to solder three wires to `GND`, `TX`, and `RX` (refer to page 8 of the FTDI datasheet file). After that solder these three wires to corresponding **TELEM2** port cable. Note that `GND` connects to `GND`, `RX` to `TX`, and `TX` to `RX`.

Plug in the DC power cable to the Odroid and check if it's powered.

Feeding data

For robot to get data from Mocap we need republish the data form vprn node to mavros vision topic by `relay` command.

First run **vrpn_client_node** on your Odroid.

Next, you will need to run MAVROS node in a new terminal on Odroid

```
roslaunch mavros px4.launch fcu_url:=/dev/ttyUSB0:921600 gcs_url:=udp://@192.168.0.
↪105:14550
```

where `fcu_url` is the serial port that connects ODROID to the flight controller. Use `ls /dev/ttyUSB*` command on your Odroid to see if serial port is connected. Parameter `gcs_url:=udp://@192.168.0.119:14550` is used to allow you to receive data to **QGroundControl** on your machine (that has to be connected to the same WiFi router). Adjust the IP to match your PC IP, that runs **QGroundControl**.

MAVROS provides a plugin to relay pose data published on `/mavros/vision_pose/pose` to PX4. Assuming that MAVROS is running, you just need to remap the pose topic that you get from Mocap `/vrpn_client_node/<rigid_body_name>/pose` directly to `/mavros/vision_pose/pose`.

```
rosrun topic_tools relay /vrpn_client_node/<rigid_body_name>/pose /mavros/vision_pose/
↪pose
```

Stream over MAVLink and check the MAVLink inspector with **QGroundControl**, the local pose topic should be in NED. Move the robot around by hand and see if the estimated local position is consistent (always in NED).

9.5 Navigating rover

9.5.1 Intro

Now it's time to control rover from the joystick.

We will need a computer running Ubuntu with Joystick connected to it. To establish Odroid communication with that computer, we will setup ROS Network. The Odroid on the rover will be the ROS Master. The joystick commands will be converted to position setpoints. The difference between rover own position and the goal position will be error for the PID controller. The output of PID controller will be published to `mavros/rc/override` topic and "simulate" the transmitter sticks movements.

9.5.2 Setup a ROS Network

First let's tell computer running Ubuntu that Odroid is the **Master** in the ROS network by editing `.bashrc` file. Open terminal and open `.bashrc` file for editing on the computer with joystick.

```
gedit ~/.bashrc
```

Add following lines to the end of the file. Just change last numbers to corresponding IP numbers.

```
export ROS_MASTER_URI=http://192.168.0.odroid_ip_number:11311
export ROS_HOSTNAME=192.168.0.computer_ip_number
```

Make sure you **source** the `.bashrc` file after this.

From computer *ssh* into an ODROID to get access to a command-line over a network. We will setup an Odroid as a Master now.

```
ssh odroid@192.168.0.odroid_ip_number
```

It will prompt to enter password, if you use minimal image provided then it's **odroid**.

Let's edit `.bashrc` file on ODROID as well.

```
nano .bashrc
```

- Add the following lines to the end of the file. Just change last numbers to corresponding IP numbers.

```
export ROS_MASTER_URI=http://192.168.0.odroid_ip_number:11311
export ROS_HOSTNAME=192.168.0.odroid_ip_number
```

To save file, press Ctrl+X, press Y, hit Enter. Source the `.bashrc` file.

9.5.3 ODROID commands

Run on Odroid separate terminals **vrpn_client_ros**, **MAVROS** and **relay**.

```
roslaunch vrpn_client_ros sample.launch server:=192.168.0.101
```

```
roslaunch mavros px4.launch fcu_url:=/dev/ttyUSB0:921600 gcs_url:=udp://@192.168.0.pc_
↪ip_number:14550
```

```
rosrun topic_tools relay /vrpn_client_node/<rigid_body_name>/pose /mavros/vision_pose/
↪pose
```

9.5.4 Computer commands

It's important at this stage to check if data from Mocap is published to `/mavros/vision_pose/pose` and `/mavros/local_position/pose` by echo'ing these topics on the computer.

Download `rover_joystick.launch` and `rover.py` files to the computer and put them into `scripts` and `launch` folder accordingly. Try to go through the code and understand what it does.

```
# Inside the scripts folder of your package
wget https://raw.githubusercontent.com/risckaust/risc-documentations/master/src/rover-
↪joystick/rover.py
chmod +x rover.py

#Inside the launch folder of your package
wget https://raw.githubusercontent.com/risckaust/risc-documentations/master/src/rover-
↪joystick/rover_joystick.launch
```

To give permission to the joystick, execute the following command and disconnect/connect joystick after this.

```
sudo chmod a+rw /dev/input/js0
```

Now run in a new terminal on the computer your launch file

```
roslaunch mypackage rover_joystick.launch
```

9.5.5 Joystick control

Try to move joystick, rover should move accordingly.

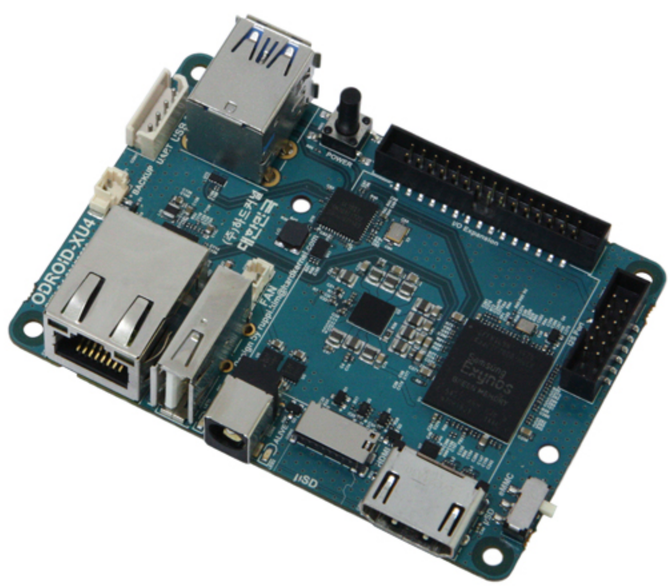
9.5.6 Contributors

Mohammad Albeaik and Kuat Telegenov.

CHAPTER 10

Companion Computers Setup

10.1 ODROID XU4 setup



10.1.1 List of components

- ODROID XU4.
- 16GB (or more) eMMc module.
- eMMC reader.
- Micro-SD reader.

- WiFi Module 3 (2.4Ghz only) good for outdoor use, or WiFi Module 5 dual band 2.4/5Ghz good for high-bandwidth.
- DC plug cable - for onboard/portable power connection.

10.1.2 Setup Ubuntu

ODROID XU 4 supports both Ubuntu and Android, see the details on official odroid webpage.

Here we will discuss how to setup Ubuntu 16.04

Flashing Ubuntu image

You can use either an SD card or eMMC. eMMC is recommended as it is much faster than SD card, 16GB or more is recommended.

For our applications we use minimal image (minimal, Bare OS) without GUI. Minimal image will have much smaller size and faster boot and less overhead in general. Extract the downloaded image from official odroid webpage and use `Etcher` to flash it to either SD or eMMC card.

User account setup

After downloading and flashing image to odroid, it is recommended to setup a user account for easier handling in the future. Plug eMMC to the odroid, and connect it to the monitor. Login using the root account (user: `root`, password: `odroid`).

```
adduser odroid # create new user with name odroid
adduser odroid sudo # add odroid user to admin group
adduser odroid dialout # give odroid user access to serial ports
```

10.1.3 Network Setup

It is recommended that you use static IP address if you plan to use ODROID via a WiFi network. This will reduce latency over wifi.

To set a static IP address on odroid, do the following.

Open `/etc/network/interfaces` file for editing by running following commmand

```
sudo nano /etc/network/interfaces
```

Add or edit the following lines

```
auto wlan0
# the following will auto-start connection after boot
allow-hotplug wlan0
iface wlan0 inet static
address 192.168.0.xxx # choose a static IP, usually you change the last number only␣
↪for different devices
netmask 255.255.255.0
broadcast 192.168.0.255
gateway 192.168.0.1 # your router IP
dns-nameservers 8.8.8.8
wpa-ssid "wifi_name"
wpa-psk "wifi_password"
```

Note: You will need modify `wlan0` to match the wifi card number on your odroid once the wifi device is connected. Is possible that it changes when you change the wifi device.

To check your wifi card number,

```
ifconfig -a
```

After odroid is connected to WiFi network and internet run the following commands

```
apt-get update
apt-get upgrade
```

Reboot the odroid and now login with newly created user.

10.1.4 Installing packages

Install ROS

To install ROS on ODROID follow official instructions from ROS wiki page. We assume that ROS Kinetic is used.

Important: Install the ROS-Base: (Bare Bones) not the full desktop version

After installing ROS, you can install ROS packages that you need individually either by using `apt-get` or from source.

Install MAVROS

```
sudo apt-get install ros-kinetic-mavros ros-kinetic-mavros-extras
wget https://raw.githubusercontent.com/mavlink/mavros/master/mavros/scripts/install_
↪geographiclib_datasets.sh
./install_geographiclib_datasets.sh # might require sudo
```

Install vrpn

```
sudo apt-get install ros-kinetic-vrpn-client-ros
```

10.1.5 Backup image

After you installed all the packages and software you might want to create an image of the entire eMMC. Plug it into the another Ubuntu running computer and execute the following comands:

```
lsbkl # Will lists the block devices
dd if=/dev/sdc of=/path_to_the_folder/backup.img # Match sdc to the eMMC from
↪previous command
# It will take time to create an image, and will create a file with full capacity of
↪the eMMC
# To reduce the size and shrink the unused space run the following
xz -c backup.img > backup.img.xz
```

10.2 Intel Up Board

- Up board is used in the Intel Realsense development kit.
- Follow this guide to setup the Up board

10.2.1 Using Edimax AC600 Wifi module

You will need to install drivers as follows:

```
sudo apt-get update
git clone https://github.com/gnab/rtl8812au.git
cd ~/rtl8812au
make
sudo make install
sudo modprobe 8812au
```

Then, reboot

Note: To be able to use `ssh` from a remote computer, you will need, `sudo apt-get install openssh-server && openssh-client`

10.3 Raspberry Pi Setup

Note: To be done.

10.4 Intel NUC setup

Note: To be done.

CHAPTER 11

Pixhawk Interface Setup

11.1 Intro

OFFBOARD control means that we would like to be able to send (usually) high-level control commands to *Pixahwk*. For example, sending position, velocity , or acceleration set-points. Then, *Pixhawk* will receive those set-points and perform the neccessary low-level control (e.g. attitude/engines control).

In general, sending high-level commands is done off-board (board here refers to *Pixhawk*). In other words, an offboard computer is usually used to execute some code to take some high-level decisions. Then, high-level decisions are translated to set-points (e.g. position set-points) which, then, are sent to the *Pixhawk* to be executed. For example, an offboard computer can be used to do run some image processing algorithm for object tracking. The output of the algorithm is position set-points to tell *Pixhawk* to move to the direction of the tracked object.

In general, executing such offboard tasks are not feasible due to the limited resources on *Pixhawk*. Therefore, more powerful computers are used.

Offboard computers can be single board computer (or SBC in brief), e.g. `ODROID XU4`. Or, it can be a fully loaded workstaton, desktop, or laptop.

In summary, *Pixhawk* is used as a flight controllers. Whereas, offboard controller are used to execute more sophisticated tasks.

In this guide, we will learn how to do offboard control from an SBC (ODROID XU4), and from desktop/laptop that runs *MATLAB*. In both cases, we need to setup the required hardware interface. We will discuss two main interfaces: Serial interface, and WiFi interface. See next sub-sections for details.

11.2 Off-board serial interface

Serial interface with Pixhawk can be done using:

- Radio modules:
  - XBee module
  - 3DR telemetry module

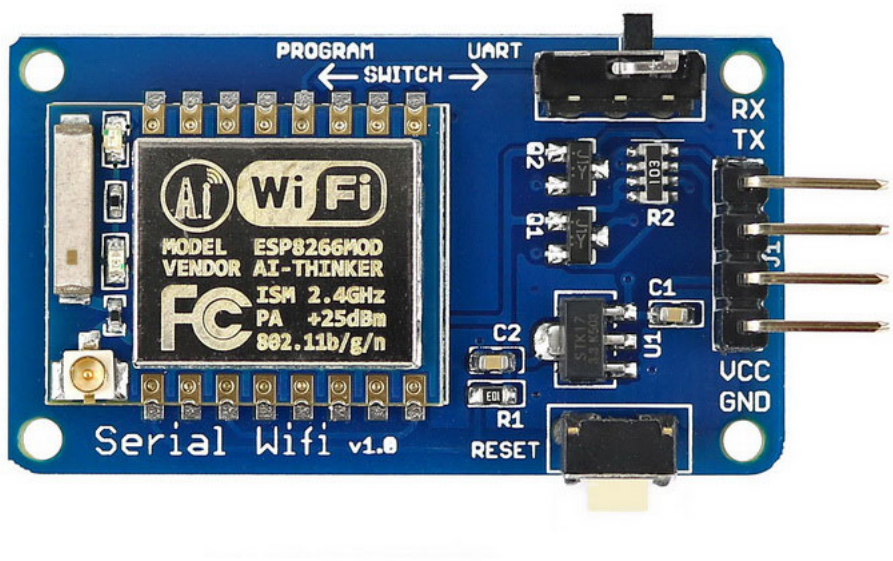
- Wired serial:
  - Direct serial interface
  - FTDI/USB

11.3 WiFi Interface with ESP-07

In this tutorial, we are going to use the `ESP8266` WiFi module to communicate with *Pixhawk* via WiFi.

Required:

- ESP-07 ESP8266 Serial Wi-Fi Wireless Transceiver Module



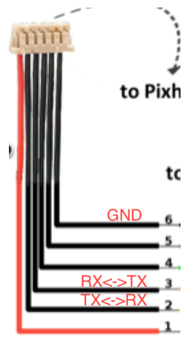
- FTDI/USB cable to flash firmware.

Connect the FTDI/USB cable to the ESP module. The orange cable (TX) is connected to (RX) pin on the module. Yellow cable (RX) is connected to (TX) pin on the module. Connect the power (red) and ground (black).

Follow the this guide to flash and setup the ESP8266.

Note: Use `platformio run -e esp01_1m -t upload` to upload the firmware to the board.

Connect the module to the Pixhawk as follows.



Important: You first need to make sure that you configured `TELEM2` port to be used for ESP link with baud 921600. You can do this, by first, connecting to Pixhawk via USB, and modify the `SYS_COMP` parameter in the *System* tab on the left. Now, you can proceed.

Power-on the Pixhawk with the WiFi module connected to `TELEM2` as mentioned above.

Search for the **Pixracer** WiFi network. Connect to that network with the password **pixracer**.

Open QGroundControl and connect using UDP connection.

Now you are connected to Pixhawk via WiFi. The Wifi Module is in *Access point* mode by default, and it creates its own WiFi network (**Pixracer**). If you wish to connect to your own local WiFi network, then in QGC, while you are connected to *Pixracer* network, go to the **WiFi Bridge** tab on the left and choose *station mode*.

Write the desired network name and password in the appropriate field.

Restart Pixhawk, and the WiFi module should try to connect to your local network.

Now, you can connect your machine to the same local network, then connect to Pixhawk from QGroundControl via UDP.

11.4 WiFi Interface with WiFly RN XV

In this section, we learn how to setup a WiFi communication with *Pixhawk* using the *RN-XV WiFly* module.

Requirements:

- *Pixhawk*: calibrated and ready to fly
- WiFi module [RN-XV WiFly Module - Wire Antenna. Available here.
- XBee explorer USB to configure WiFi module via PC
- Xbee breakout board to interface with *Pixhawk*

In this tutorial, `TELEM2` is going to be used to connect the WiFi module at baud rate 921600. `TELEM1` can be used too, but will require further configuration steps, but you can still use it directly at baud `57600` (which is its default).

11.4.1 Pixhawk TELEM setup

To set the baude rate of `TELEM2` to `921600`, connect *Pixhawk* to *QGroundcontrol*. Go to the *System* tab. Change the `SYS_COMP` parameter to use companion with 921600 baudrate. Restart *Pixhawk* to take effect.

11.4.2 WiFi module setup

Official Roving Network documentation

Connect the WiFi module to the XBee explorer USB board and connect it to the computer. You will need to use a serial terminal. For Mac, use the Mac terminal. For Windows it is recommended to use **TeraTerm**.

On a Mac terminal, use the screen command to log into the Wifly

```
screen /dev/tty.usbserial-FTFABC 9600 8N1
```

`/dev/tty.usbserial-FTFABC` is the device port on Mac. You can find yours using

```
ls /dev/tty*
```

After you login, type **$$$** and hit **ENTER**

Type to make sure that the device is operational.

```
scan
```

If there are networks, it should be listed.

11.4.3 Serial setup

You can change the serial baudrate by

```
set u b 57600
```

Warning: Make sure that you use the new baud rate to connect again to the device via serial port.

11.4.4 WiFi setup

Set authentication to WPA2-PSK only:

```
set wlan auth 3
```

Set auto channel scan

```
set wlan channel 0
```

Tell the module to auto-join the network when powered on:

```
set wlan join 1
```

Set wireless name, SSID

```
set wlan ssid <your wifi ssid>
```

Set WiFi password

```
set wlan phrase <password>
```

Enable continous scanning

```
set wlan linkmon 5
```

11.4.5 IP setup

This guide assumes UDP communication to a ground control station computer on IP `192.168.1.100`, port `14550` (QGroundControl default port).

Set dynamic IP (recommended)

Enable DHCP on each boot (for dynamic IP):

```
set ip dhcp 1
```

Set IP protocol (UDP & TCP)

```
set ip protocol 3
```

Set remote port:

```
set ip remote 14550
```

Set remote hos IP (IP of your PC):

```
set ip host 192.168.1.100
```

Test and save configurations

Join the WiFi

```
join <WiFi ssid>
```

If it connects, it will show:

Save and reboot

```
save
reboot
```

> **Attention:** Make sure that you save your settings, otherwise it will be lost

To check the settings current yon the device,

- IP settings:

```
get ip
```

- WiFi settings:

```
get wlan
```

- Serial settings:

```
get u
```

Static IP

Disable DHCP mode

```
set ip dhcp 0
```

Set the WiFi module's IP address

```
set ip address <choose ip>
```

your IP first 3 numbers (e.g. `192.168.1.\*`) should be the same as your router's first three numbers

Set IP gateway (usually this is your router's IP). You can first set up dynamic IP, and then connect to the WiFi. Then, on the WiFi module command line type `get ip` to see the *gateway* and the *netmask*, and note them down. Set the *gateway* and *netmask* as follows,

```
set ip gateway <router ip address>
```

Set *netmask*:

```
set ip netmask <netmask address>
```

Set local port. You can leave the default (2000)

```
set ip localport 2000
```

Set the remote host IP and remote port as before.

Save and reboot

```
save
reboot
```

Make sure that the device can join the WiFi network. Log in to the device using (e.g. `screen` command), and type **$$$**. Then join the network by typing join <network ssid>

Once successful, you can now go to next step to set higher baud rates.

Configure higher baud rates

> **Warning:** DO NOT set high baud rates while you are on serial (e.g. `921600`), because you will not be able to log in again from the serial console. You can set higher baud rate after you log in to the WiFly module via WiFi, using `telnet` command in Mac OS

First make sure your computer is connected to the same router as the WiFly device. Open a terminal and type,

```
telnet <wifly ip address> <wifly localport>
```

then type `$$$`, and hit **ENTER**

Set high baudrate

```
set u b 921600
```

Save and reboot

```
save
reboot
```

Finally, attach the WiFly device to an XBee explorer requlated board, and connect it to `TELEM2`.

Now you are ready to communicate with the *Pixhawk* via WiFi!

CHAPTER 12

ODROID to MATLAB Stream

12.1 Intro

Required:

- ODROID with OpenCV installed.
- Computer with OpenCV installed in default locations.
- MATLAB with associated compiler e.g. XCode(Mac OS)/Visual Studio or Microsoft SDK (for Windows)
- WiFi network (Access Point)
- Streaming ODROID application and MATLAB receiving application.

12.2 ODROID setup

12.2.1 Setup OpenCV

Make sure that your odroid is connected to internet.

Open a terminal window, and run the following command,

```
sudo apt-get -y install libopencv-dev

sudo apt-get -y install build-essential cmake git libgtk2.0-dev pkg-config libavcodec-
↪dev libavformat-dev libswscale-dev python-dev python-numpy libtbb2 libtbb-dev␣
↪libjpeg-dev libpng-dev libtiff-dev libjasper-dev libdc1394-22-dev
```

12.2.2 Setup streaming app

Create a clean directory and navigate to it e.g.

```
cd ~/Desktop
mkdir imgstream
cd imgstream
```

Clone the streaming app from Github

```
git clone https://github.com/mzahana/Image_Live_Stream.git
cd Image_Live_Stream
```

Navigate to the `stream_cpp` folder, and compile the app

```
cd opencv_stream/stream_cpp
cmake . & make
```

If all goes well, then two executable files should be generated: `sender` and `receiver`. Otherwise, make sure that you installed OpenCV properly in the default locations.

To stream images over network, use the `sender` app after you connect a camera to ODROID. To use the `sender` app, use the following command in a terminal, inside the `stream_cpp` folder,

```
./sender 192.168.1.100 10000
```

where `192.168.1.100` is the IP of machine running MATLAB (the host machine) (which should be on the same network as the ODROID's). `10000` is the port that MATLAB is listening on. Use appropriate IP and port that match the host ones.

12.3 MATLAB setup

12.3.1 On MacOS

Make sure that you installed XCode on your Mac OS.

Make sure that you associat your MATLAB with XCode compiler (Google it). Run `mex -setup` in MATLAB command line for more information.

Navigate to the `Image_Live_Stream` folder that you downloaded from Github.

Run the `setup.m` file

```
>> setup
```

If all goes well, you are ready to receive live stream of images from ODROID.

Look at the `testScript.m` file to see how you can use the `ImgStream` class to establish the connection, and receive image data.

12.3.2 On Windows

Make sure that you install OpenCV 2.4.13 on your Windows. Follow this video. It is assumed that you installed the OpenCV folder in `C:\`

Make sure that your MATLAB is associated with compiler. Run `mex -setup` in MATLAB command line for more information.

In MATLAB, run the `setup.m` file.

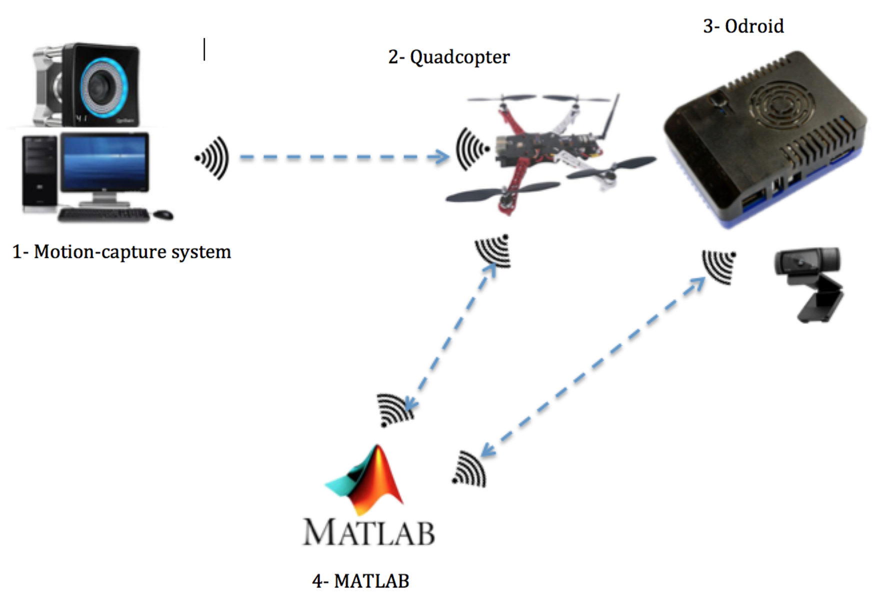
If all goes well, you are ready to receive image stream. Look at the test script to get familiar on how to use the `ImgStream` Class.

CHAPTER 13

MATLAB Pixhawk Communication

In this demo you will learn two things

- Sending high-level commands from MATLAB to Pixhawk, using `MATMAV`
- Getting live stream of images into MATLAB from ODROID which is mounted on a quadcopter.



As you can see from the previous figure, there are 4 main components to setup.

- Motion capture system.
- Quadcopter with Pixhawk flight controller.
- ODROID: embedded Linux computer.
- MATLAB enviornment.

13.1 Motion capture setup

Motion capture (or Mocap in short) is used to provide accurate positions and orientations in an indoor environment. The mocap setup we have in the lab is from *Optitrack* company. You can think of it as GPS system for indoor environment.

Mocap mainly consists of cameras, network switches, and a PC with a special software. Cameras capture images which contain special *reflective markers*. Those markers are used to track objects (rigid bodies) they are attached to. Then, images from all cameras are transmitted to the PC software (called *Motive*) through the network switches, in order to do further image processing.

Motive extracts useful information about captured rigid bodies such as position and orientation. Such information can be further transmitted through network to other PCs for further usage. Rigid bodies are defined by at least 3 reflective markers that are rigidly mounted on the object of interest.

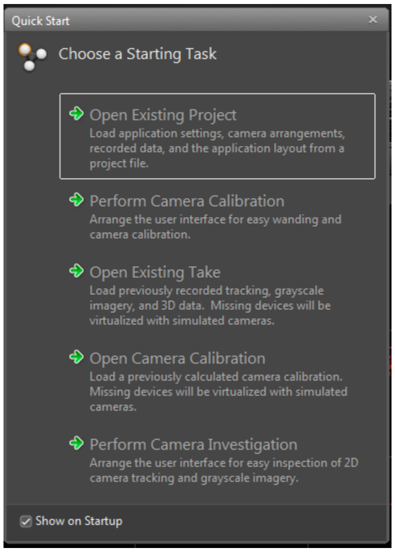
For this tutorial, it is assumed that the Mocap is already calibrated.

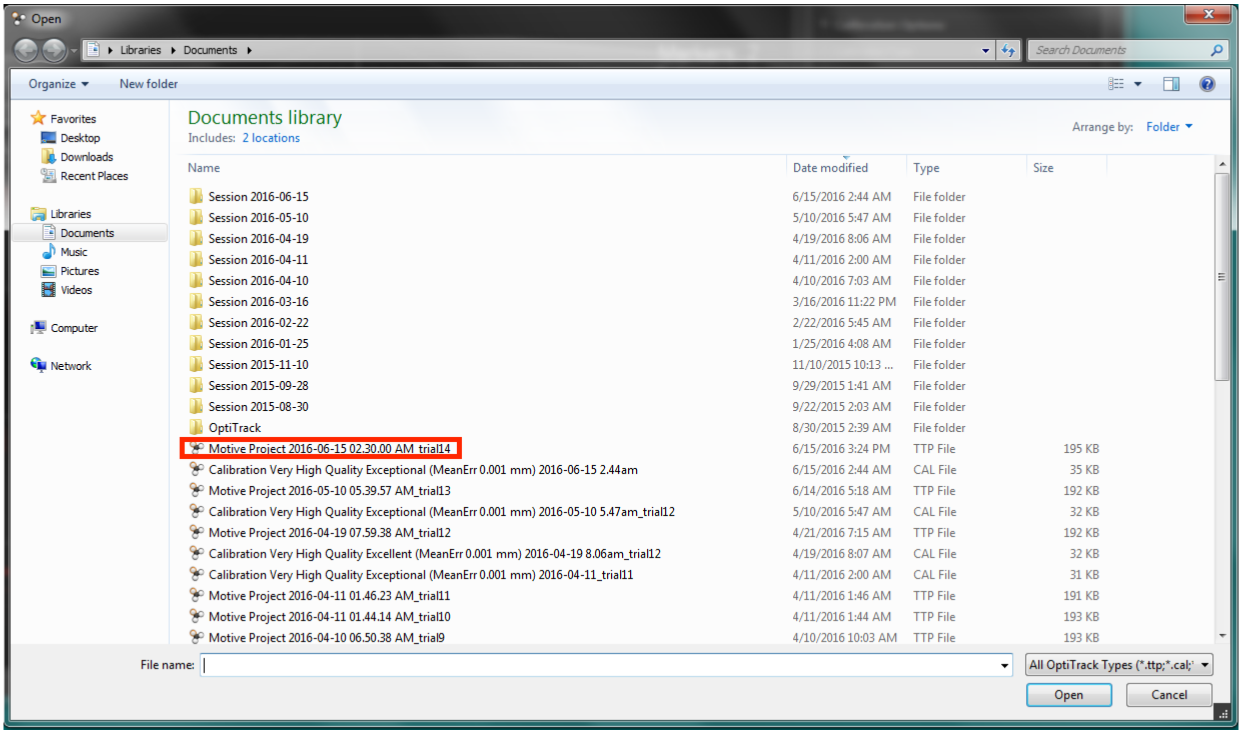
What we need in this tutorial is to

- Open *Motive* project
- Define rigid bodies
- Configure streaming parameters in Motive
- Use the Streaming Application to send mocap info to Pixhawk

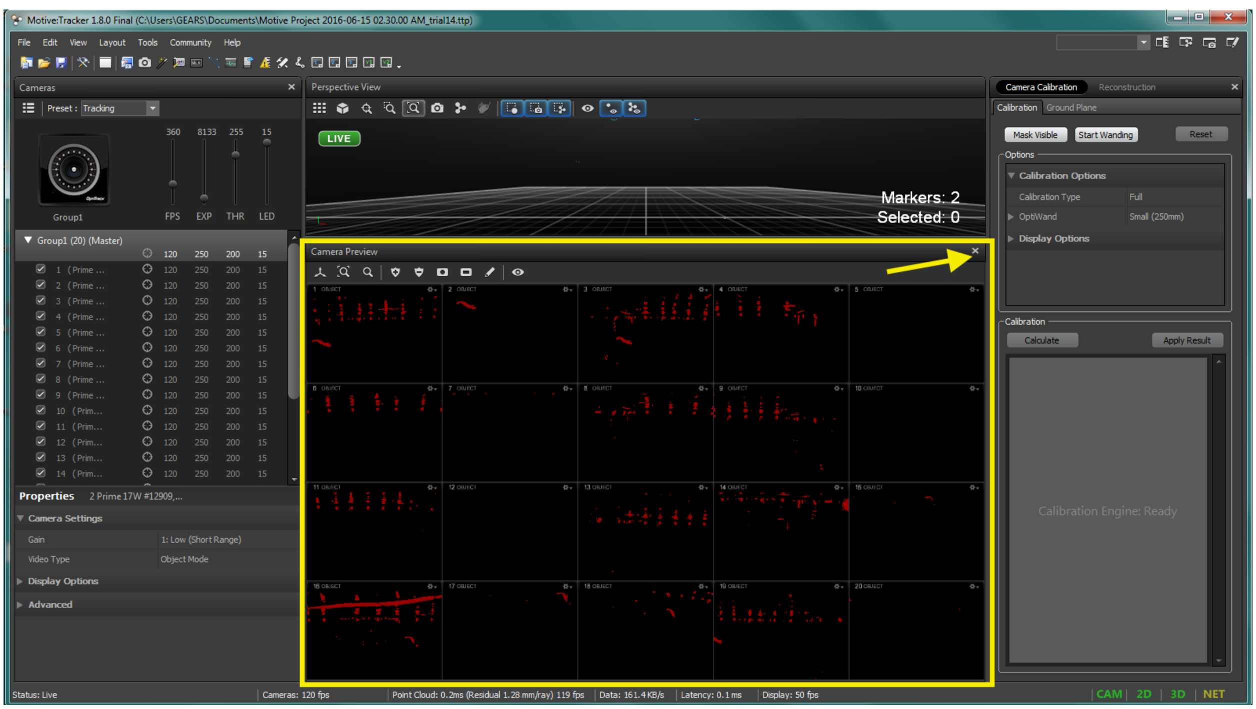
Follow the following steps in order.

- Open *Motive* software, and choose `Open Existing Project`. Choose a recent project that represents the latest calibration settings.

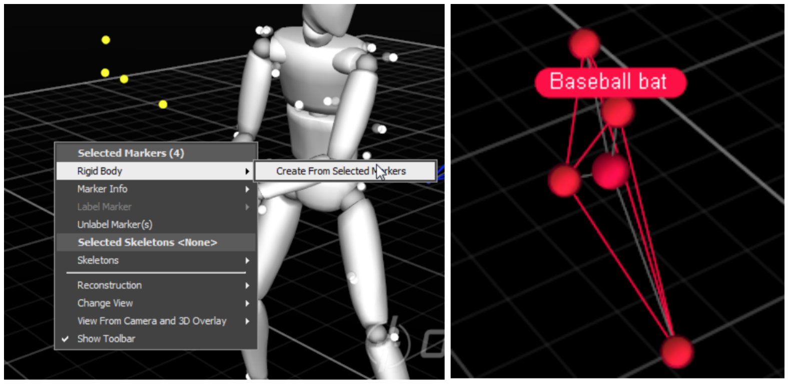




- Close the `Camera Preview` view, and leave the `Perspective View` view for 3D viewing of objects.

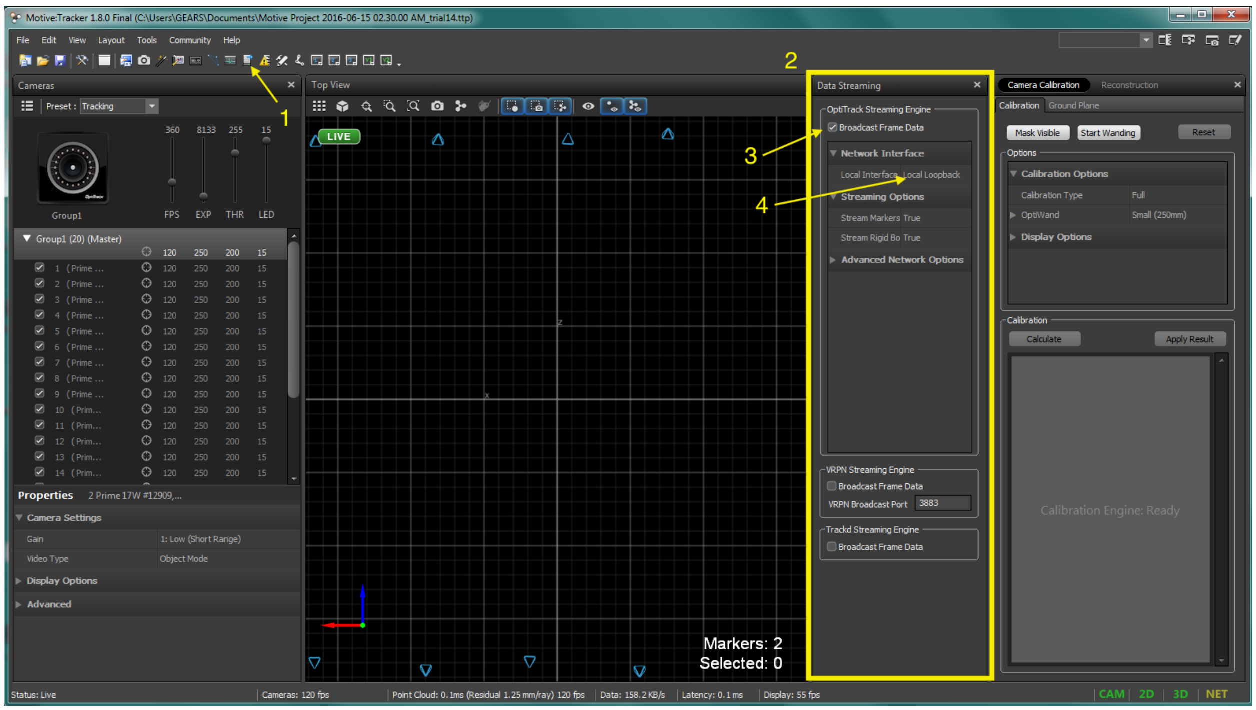


- Place the object in the cage (e.g. quadcopter) with mounted markers (minimum 3 markers).
- Select markers in the `Perspective View` and create a rigid body

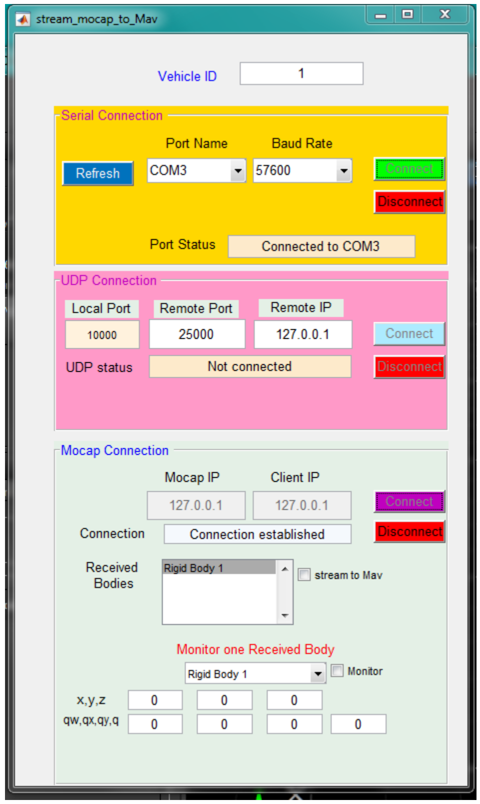


Creating a rigid body from the perspective view

- You can know your rigid body number from the `Rigid Body`, after you select the rigid body in the `Perspective View`.
- Now, activate streaming over network as follows



- Connect the wireless serial module to the Mocap PC (e.g. XBee)
- Open Mocap streaming App.



- Select the proper *Vehicle ID*
- In the `Serial Connection` tab, select the proper serial port of the communication module from the `Port Name` drop menu. Set the `Baud Rate` to *57600*. Finally, click the `Connect` button. If the connection is successful, it will show a status message in the `Port Status` field.
- In the `Mocap Connection` tab, leave the `Mocap IP` and `Client IP` to the defaults IPs (`127.0.0.1`). Hit the `Connect` button.
- If the connection is successful, you should see the defined rigid bodies in the `Received Bodies` list box.
- Select the one corresponds to the quadcopter. Then, check the `stream to Mav` checkbox.
- Now, your quad should be getting its position and orientation feedback from the Mocap system.

13.2 Quadcopter setup

This tutorial assumes that the quadcopter is setup and equipped with a calibrated Pixhawk (or Pixracer) flight controller.

In this Demo, the quadcopter is assumed to have an ODROID on-board, two serial communication modules (e.g. XBee). One for the Mocap connection, and the other for MATLAB connection.

13.3 ODROID setup

In this Demo, ODROID is used to capture real-time images and stream them over WiFi network to a MATLAB session. The streaming application is assumed to be installed on ODROID and ready to be used. Also, the ODROID is assumed to be setup to connect to a local WiFi network.

Check this guide to see how to install the streaming app on ODROID.

To run the application, follow the following steps in order

- Connect a compatible camera to ODROID
- Connect a compatible WiFi module to ODROID (use the ODROID WiFi adapter)
- Power on the ODROID
- From your laptop (which is connected to the same local WiFi network as the ODROID), open a terminal and remotely log-in to ODROID

```
ssh odroid@192.168.1.113
password: odroid
```

`odroid` is the user account name. `192.168.1.113` is the ODROID's IP address.

- Navigate to the app folder and run it

```
cd ~/Desktop/imgstream/Image_Live_Stream/opencv_stream/stream_cpp
./sender 192.168.1.112 10000
```

`192.168.1.112` is your machine's IP address. `10000` is the port that is going to be opened in your MATLAB. You can choose another port, but make sure it matches the one used in your MATLAB.

- Now, the ODROID is sending images to the specified IP and port.

13.4 MATLAB setup

In this Demo, MATLAB is used to

- Communicate with Pixhawk (or Pixracer) in order to send high-level commands. For example, position set-points, velocity set-points, or acceleration set-points. It can also receive feedback information from Pixhawk.
- Receive live-stream of images from ODROID.

> **Warning:** You need to use the MATLAB files associated with this Demo. Please ask for your free copy.

We are going to use two main MATLAB classes in this Demo. One is called `MatMav`, and the other is called `ImgStream`.

`MatMav` is a MATLAB class that is used to communicate with Pixhawk. `ImgStream` is a MATLAB class that is use to receive live image stream from ODROID (or any Linux computer) over network.

Before you use the MATLAB files associated with this demo, you should setup your environment properly.

> **Warning:** Before you use the MATLAB files associated with this demo, you should setup your environment properly. Namely, you need to associate your MATLAB with a C/C++ compiler, and install OpenCV.

Please follow the OpenCV installation as follows,

- For Mac OS.
- For Windows.

Google how to associate your MATLAB with a compiler.

- Download the MATLAB folder associated with this Demo.
- Open MATLAB and navigate to that folder.
- Run the `setup.m` file.

If all goes well, you should get the message `Setup is done`. Now, you are ready to proceed with the experiment which is implemented in the `Demo1.m` MATLAB file.

- Check the `Demo1.m` file to get familiar with `MatMav` and `ImgStream` classes.

CHAPTER 14

Setup HIL with PX4 & V-REP

Warning: This is an old attempt to do hardware-in-the-loop simulation with VREP simulator and is deprecated. The current ongoing work is to make HIL with Gazebo via a MAVROS plugin. See following link. HIL with Gazebo via a MAVROS plugin in progress.

14.1 Prerequisites

- Machine with Ubuntu 14.04 LTS installed
- ROS Indigo installed
- CATKIN workspace
- V-REP installed
- V-REP HIL scene: can be found in the `catkin_ws/src/v_repExtRosInterface/vrep_hil` folder after you run the setup `.sh` script.
- Pixhawk loaded with PX4 HIL firmware.
- Customized .params file for appropriate Pixhawk parameters configurations: can be found in the `catkin_ws/src/v_repExtRosInterface/vrep_hil` folder after you run the setup .sh script
- Setup script `vrep_px4_hil_setup.sh` (see the code below)
- Internet connection

14.2 Setup

Prepare the setup file as described in the below section.

Open a new terminal, navigate to the setup file, and define the setup variables: `VREP_ROOT` is the VREP main folder's path, `ROS_WORKSPACE` is the path to your catkin workspace. Finally, run the setup script (see the code below). **Make sure you have internet connection and root access via** `sudo`.

```
# export VREP_ROOT=path/to/vrep/folder
# export ROS_WORKSPACE=path/to/catkin/workspace
./vrep_px4_hil_setup.sh
```

Once the installation is successful, connect Pixhawk via USB. Run `mavros` to connect to Pixhawk,

```
roslaunch mavros px4.launch fcu_url:=/dev/ttyACM0:115200 gcs_url:=udp://@192.168.1.135
```

You may need to adjust `fcu_url` address `/dev/ttyACM0:115200` and `gcs_url` address `udp://@192.168.1.135` according to your setup.

In another terminal, run V-REP: Navigate to VREP main folder, then execute

```
# cd /vrep/folder
./vrep.sh
```

Load the `px4_hil.ttt` scene, and run it. You should see the main LED on Pixhawk go green. It means it's able to get xyz data (fake GPS).

14.3 Setup Shell Script

You can create the setup file by copying the following shell code to a file, and then, run it. Make sure it has `.sh` extension, and make it executable : `chmod +x <filename.sh>`.

```
#!/bin/bash

# Check if required environment variables are set properly
if [ ! -v ROS_WORKSPACE ]; then
    echo "!!!! ERROR: ROS_WORKSPACE is unset"
    echo "set it using: export ROS_WORKSPACE=path/to/workspace/folder"
    echo "press 'ENTER' to exit....."
    read x
    exit 1
fi

if [ ! -v VREP_ROOT ]; then
    echo "!!!! ERROR: VREP_ROOT is unset"
    echo "set it using: export VREP_ROOT=path/to/VREP/folder"
    echo "press 'ENTER' to exit....."
    read x
    exit 1
fi

ROS_WORKSPACE1=$(echo $ROS_WORKSPACE | tr -d '\r')
VREP_ROOT1=$(echo $VREP_ROOT | tr -d '\r')

# Clean ros_ws: build/devel/logs directories
cd $ROS_WORKSPACE1
rm -r -f devel/
rm -r -f build/
rm -r -f logs/
```

(continues on next page)

(continued from previous page)

```
# Initialize catkin workspace
cd "$ROS_WORKSPACE1"
catkin init
cd src
rm .rosinstall
cd ..
wstool init src

# Remove old vrep ros interface package`
cd "$ROS_WORKSPACE1/src"
if [ -d "v_repExtRosInterface" ]; then
    rm -r -f "$ROS_WORKSPACE1/src/v_repExtRosInterface"
fi

# Remove old mavros package
# remove mavros if installed by apt-get
sudo apt-get remove ros-indigo-mavros
sudo apt-get remove ros-indigo-mavros-extras
sudo apt-get remove ros-indigo-mavros-msgs
if [ -d "mavros" ]; then
    rm -r -f mavros
fi

# Remove mavlink package
# remove mavlink if installed by apt-get
sudo apt-get remove ros-indigo-mavlink
if [ -d "mavlink" ]; then
    rm -r -f mavlink
fi

# Create Python-packages folder,
cd ~
# check if directory exists
if [ ! -d "python-packages" ]; then
    mkdir -p "python-packages/src"
fi

# Get some required python packages
sudo apt-get update
sudo apt-get install python-tempita python-catkin-tools python-rosinstall-generator␣
↪python-pip -y
sudo pip install future

# Clone fresh vrep ros interface package
cd "${ROS_WORKSPACE1}/src"
git clone https://github.com/mzahana/v_repExtRosInterface.git
# Copy some V-REP packages from V-REP folder
cp -R "${VREP_ROOT1}/programming/ros_packages/vrep_common/" "${ROS_WORKSPACE1}/src/"
cp -R "${VREP_ROOT1}/programming/ros_packages/vrep_joy/" "${ROS_WORKSPACE1}/src/"
cp -R "${VREP_ROOT1}/programming/ros_packages/vrep_plugin/" "${ROS_WORKSPACE1}/src/"
cp -R "${VREP_ROOT1}/programming/ros_packages/vrep_plugin_skeleton/" "${ROS_
↪WORKSPACE1}/src/"
cp -R "${VREP_ROOT1}/programming/ros_packages/vrep_skeleton_msg_and_srv/" "${ROS_
↪WORKSPACE1}/src/"

# Get fresh mavros package
git clone https://github.com/mzahana/mavros.git
```

(continues on next page)

(continued from previous page)

```
# checkout the px4_hil_plugins branch
cd mavros
git checkout  px4_hil_plugins
cd "${ROS_WORKSPACE1}"

# Get fresh mavlink package
rosinstall_generator --rosdistro kinetic mavlink | tee /tmp/mavros.rosinstall
wstool merge -t src /tmp/mavros.rosinstall
wstool update -t src -j4
rosdep install --from-paths src --ignore-src -y

# Get supporting package for vrep ros interface
cd ~/python-packages
# Remove old package if exists
if [ -d "v_repStubsGen" ]; then
    rm -r -f v_repStubsGen
fi

# Get a fresh copy of the supporting python package
git clone https://github.com/fferri/v_repStubsGen.git
export PYTHONPATH=$PYTHONPATH:~/python-packages

# Build ros/catkin workspace
#VERBOSE=1 catkin build -v -p1 -j1 --no-status
#catkin build -p1 -j1
cd "${ROS_WORKSPACE1}"
catkin build

# clone built libs to V-REP folder
cp -r "${ROS_WORKSPACE1}/devel/lib/libv_repExtRosInterface.so" ${VREP_ROOT1}
cp -r "${ROS_WORKSPACE1}/devel/lib/libv_repExtRos.so" ${VREP_ROOT1}
cp -r "${ROS_WORKSPACE1}/devel/lib/libv_repExtRosSkeleton.so" ${VREP_ROOT1}
#cp -r $ROS_WORKSPACE/src/ros_bubble_rob/bin/rosBubbleRob ~/V-REP_PRO_EDU_V3_3_2_64_
↪Linux/
#cp -r $ROS_WORKSPACE/src/ros_bubble_rob2/bin/rosBubbleRob2 ~/V-REP_PRO_EDU_V3_3_2_64_
↪Linux/

source "${ROS_WORKSPACE1}/devel/setup.bash"
```

CHAPTER 15

Autostart service after system boot

Sometimes, It is more convenient to run ROS launch files automatically after robot's computer boots up. For example, if you are working with multiple drones in a swarm, it is painful to log into each drone and run mavros manually. So, to solve this issue, we can run mavros automatically after system boots. This tutorial shows you one way on how to run ROS launch file after system starts.

15.1 Create a simple systemd service

15.1.1 Use case: auto start MAVROS node

- Create a shell script and type the commands that you would execute in a normal terminal. Fro example,

```
mkdir ~/scripts
cd ~/scripts
touch startup_launch.sh
chmod +x startup_launch.sh
```

Type the following in the `startup_launch.sh` file (you can use the `nano startup_launch.sh` command). It is assumed that the username is `odroid`

```
#!/bin/bash
source /opt/ros/kinetic/setup.bash
source /home/odroid/catkin_ws/devel/setup.bash
roslaunch mavros px4.launch
```

- Create `mavros.service` file in `/lib/systemd/system`

```
cd /lib/systemd/system
sudo nano mavros.service
```

- Add the following contents:

```
[Unit]
Description=mavros

[Service]
Type=forking
ExecStart=/home/odroid/scripts/startup_launch.sh
Restart=on-failure

[Install]
WantedBy=multi-user.target
```

Save and exit by hitting `CTRL+x`, then `Y`, then `[ENTER]`

Then run:

```
sudo systemctl daemon-reload
```

And enable it on boot:

```
sudo systemctl enable mavros.service
```

Then, reboot and `px4.launch` should be executed after boot.

To disable a service,

```
sudo systemctl disable mavros.service
```

CHAPTER 16

Multi-point Telemetry

In this tutorial, the objective is to make the setup which allows to communicate with multiple telemetry modules using a single base telemetry module. The use case is a single telemetry module (e.g. 3DR or RFD900) is connected to ground station that runs QGroundControl that monitors/controls a fleet of drones. Each drone has a single telemetry module. So, it's one-to-many network.

We will need the SiK_Multipoint.

16.1 Installation

Install required packages.

On Mac,

```
brew install sdcc
```

On Ubuntu,

```
sudo apt update; sudo apt install sdcc
```

Clone the SiK package and switch to branch

```
cd ~
mkdir ~/src
cd ~/src
git clone https://github.com/RFDesign/SiK.git
cd SiK
git checkout SiK_Multipoint
```

Make and install,

```
cd SiK/Firmware
make install
```

16.2 Upload Firmware to the radio

Change the serial port name

```
tools/uploader.py --port /dev/tty.usbserial-CHANGETHIS dst/radio~hm_trp.ihx
```

Note: If you get errors, try to update `pyserial` module

16.3 Device Configuration

Start command mode,

```
screen /dev/tty.usbserial-CHANGETHIS 57600 8N1
```

then type,

```
+++
```

Wait one second before you type anything

To list all editable parameters type,

```
ATI5
```

To change a paramter use,

```
ATS<parameternumber>=<value>
```

Make sure you save by typing,

```
AT&W
```

- Set `MAVLINK=1`
- Set `NODECOUNT` to the number of used telemetry modules
- There must be a base module with `NODEID=0`
- Put base node in broadcast mode by setting `NODEDESTINATION=65535`
- All other nodes should to talk to base only by setting `NODEDESTINATION=0`

Warning: Make sure that you save parameters after each set using `AT&W`. Otherwise, parameters changes won't survive restes.

16.4 References

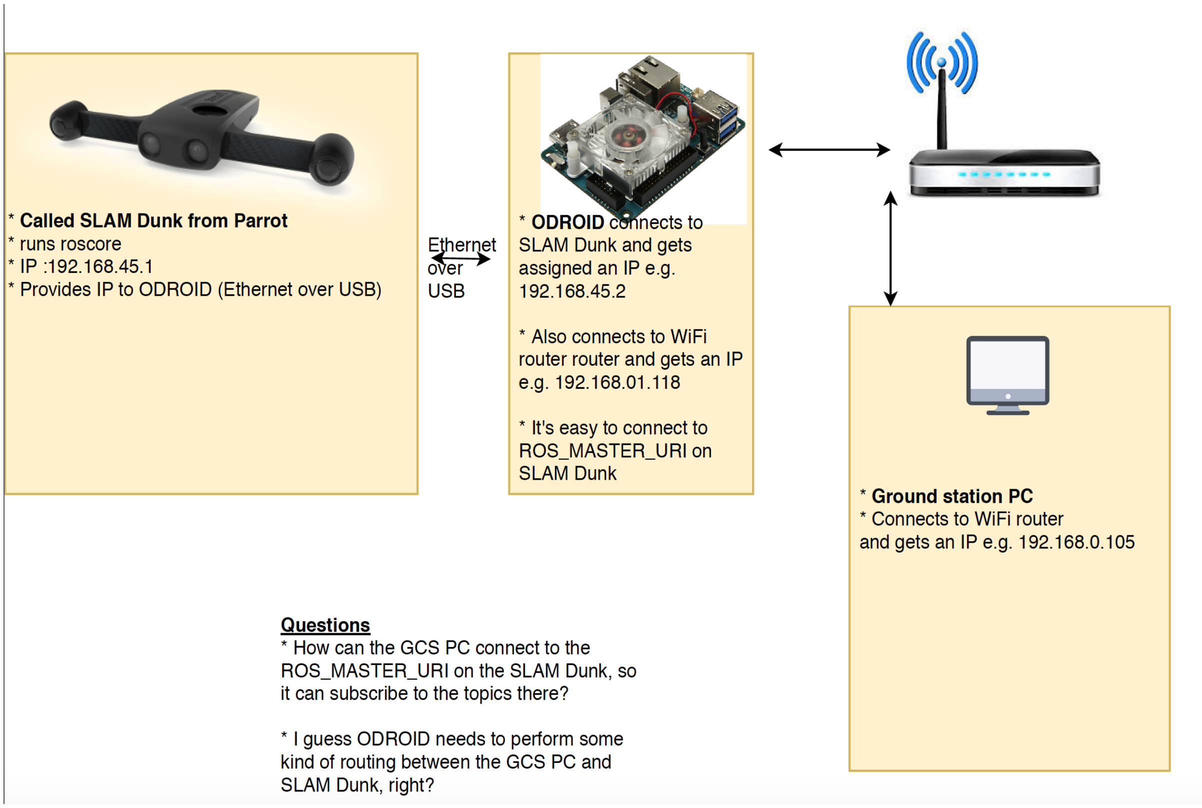
- https://github.com/RFDesign/SiK/tree/SiK_Multipoint
- http://dev.px4.io/en/data_links/sik_radio.html

CHAPTER 17

Networking

17.1 Case 1: Communication with Parrot SLAM Dunk

Consider the following setup



- the SLAM DUNK module is connected to ODROID XU4 using Ethernet over USB cable. The SLAM module has the IP `192.168.45.1`

- The SLAM module runs `roscore` and `ROS_MASTER_URI=http://192.168.45.1:11311`
- ODROID (Ubuntu 16/ROS Kinetic) detects new interface as `usb0` and get an assigned IP from SLAM module.
- ODROID also connects to a WiFi router `192.168.0.1` through an interface `wlan0` with a static IP e.g. `192.168.0.118`
- The `usb0` and the `wlan0` interfaces are independent
- There is a ground station PC that is connected to the WiFi router and has a static IP e.g. `192.168.0.105`

17.2 Summary of network devices setup

17.2.1 SLAMDUNK

- IP: `192.168.45.1`
- gateway: `192.168.45.1`
- netmask: `255.255.255.0`

17.2.2 ODROID

- IP (usb0): `192.168.45.2`
- IP (wlan0): `192.168.0.118`
- `ROS_MASTER_URI=http://192.168.45.1:11311`
- `ROS_HOSTNAME=192.168.45.2`
- Edit `/etc/hosts`, and add SLAM DUNK host name (`192.168.45.1 slamdunk-00316.local`)

17.2.3 PC

- IP: `192.168.0.105`
- gateway: `192.168.0.1`, wifi router's IP
- netmask: `255.255.255.0`
- `ROS_MASTER_URI=http://192.168.45.1:11311`
- `ROS_HOSTNAME=192.168.0.105`

17.3 IP routing

We need to route between two networks on the ODROID

- Enable ip forward on ODROID:
- in `/etc/sysctl.conf`, uncomment (or add) `net.ipv4.ip_forward=1`
- add static route on sLAM DUNK module

```
sudo route add -net 192.168.0.0 netmask 255.255.255.0 gw 192.168.45.2
```

- Add static route on PC

```
sudo route add -net 192.168.45.0 netmask 255.255.255.0 gw 192.168.0.118
```

- (Optional): On ODROID, you can modify `iptables` as follows (they will re-set after reboot)

```
sudo iptables -A FORWARD --in-interface usb0 -j ACCEPT
sudo iptables --table nat -A POSTROUTING --out-interface wlan0 -j MASQUERADE
```

Check if you can ping all devices to each other. Also, check if you can `rostopic list` and `rostopic echo` on all three devices.

17.4 To make the routing persistent

1. create a script file in the `/etc/init.d/ folder`.
2. add your route definitions to this file and change it to an executable file (`chmod +x /path/to/file`).
3. run the `update-rc.d <filename> defaults` command to make the script executable at boot time.
4. reboot the system and check whether the system adds the routes at startup(`netstat -rn`).

CHAPTER 18

DJI M100 ROS setup

Setting up the DJI M100 with on-board computer (before you do these steps you must ensure that your drone is binded with your transmitter, your drone is activated and your on-board computer is connected to the drone through the UART port. DJI hardware setup page.

- Use this link to download and install the DJI SDK to your on board computer (use instructions for ROS).
- You will have a main `dji_sdk.launch` file that runs all sdk features, you need to edit this launch file with the APP ID and APP key and the appropriate Baud rate. You might want to download the DJI Assistant app on a Windows to set some parameters like the baud rate (that needs to match the launch file) or to run simulation. To generate an APP ID and key you will need to register on the DJI website as a developer and activate your account through email. The registration process is a two steps process where in the second step you will need to provide a phone number or credit card info. The second step might require you to press on resend the activation email many times and go again in your email and activate. After you are done with the second step of registration you can now create an APP to generate an IPP ID and key.
- Your drone needs to be in "function mode" (mode is changed from transmitter) and you need to launch the main sdk launch file to start using DJI topics and services. The launch file won't launch without an appropriate APP ID and key. Also you might have an error with the drone activation so you might need to connect your transmitter to a phone that has the DJI account that you activated your drone with before running the launch file. The phone must have the DJI go App. The APP helps in calibrating sensors and receiving camera feedback (there must be a way to receive camera live streaming through ROS also)
- If you want to have permission to publish to control the DJI drone with SDK you will need to call a ROS service `/dji_sdk/sdk_control_authority` with boolean arguments (1 for authority).
- If you want to publish local position information to the `/dji_sdk/local_position` topic you will need to call the `/dji_sdk/set_local_pos_ref` service
- If you want to take-off or land you can call the `/dji_sdk/drone_task_control` with argument **4** for take-off and **6** for landing
- You can publish GPS setpoints to the `/dji_sdk/flight_control_setpoint_ENUposition_yaw` topic after converting them to **ENU**. Install `pymap3d` package by typing

```
sudo pip install pymap3d==1.6.3
```

You can import a function from there called `geodetic2enu`.

- Alternatively you can publish velocity setpoints to `/dji_sdk/flight_control_setpoint_ENUvelocity_yawrate` topic to navigate to specific GPS setpoint.

Note: You can't use velocity and position setpoints simultaneously.

- More info about topics and services here.
- A sample code for GPS navigation along with a launch file that automatically runs the DJI main node and the required services is available on the RISC Github page.

18.1 Contributors

Sarah Toonsi and Fat-hy Omar Rajab.

CHAPTER 19

DJI Manifold initial setup

If you follow official DJI User Manual for DJI Manifold you may encounter few problems. This guide will solve them and install all neccessary packages.

19.1 Enter recovery mode

Follow DJI User Manual to enter the Recover mode on DJI Manifold. Method 1 should work fine.

19.2 Restore the Manifold to default settings

```
cd
mkdir manifold
cd manifold
wget https://dl.djicdn.com/downloads/manifold/manifold_image_v1.0.tar.gz
tar -xvpzf manifold_image_v1.0.tar.gz
cd Linux_for_Tegra
sudo ./flash.sh jetson-tk1 mmcblk0p1
```

19.3 Compiling and installing kernel

Power on the Manifold and connect it to a monitor and a keyboard.

On Manifold download kernel by following commands

```
cd
mkdir ~/kernel
cd kernel
wget https://dl.djicdn.com/downloads/manifold/manifold_kernel_source_v1.0.tar.gz
```

(continues on next page)

(continued from previous page)

```
tar xvzf manifold_kernel_source_v1.0.tar.gz
cd linux_3.10
cp arch/arm/configs/manifold_config .config
make menufconfig
```

Press ESC button two times once the graphical menu will popup.

Execute to compile kernel

```
sudo make
```

You might get different errors here, mainly because of low memory. Keep executing previous command until you see it successfully done.

Next install kernel and modules

```
sudo make modules
sudo make modules_install
sudo cp /boot/zImage /boot/zImage.bak
sudo cp arch/arm/boot/zImage /boot/
sudo cp arch/arm/boot/dts/tegra124-jetson_tk1-pm375-000-c00-00.dt* /boot/
```

After this reboot Manifold

```
sudo reboot
```

19.4 Cuda and OpenCV4Tegra

```
cd
wget http://developer.download.nvidia.com/embedded/L4T/r21_Release_v3.0/cuda-repo-l4t-
↪r21.3-6-5-prod_6.5-42_armhf.deb
sudo dpkg -i cuda-repo-l4t-r21.3-6-5-prod_6.5-42_armhf.deb
sudo apt-get update
sudo apt-get install cuda-toolkit-6-5
```

```
cd
wget developer.download.nvidia.com/embedded/OpenCV/L4T_21.2/libopencv4tegra-repo_l4t-
↪r21_2.4.10.1_armhf.deb
sudo dpkg -i libopencv4tegra-repo_l4t-r21_2.4.10.1_armhf.deb
sudo apt-get update
sudo apt-get install libopencv4tegra libopencv4tegra-dev libopencv4tegra-python
```

19.5 ROS Indigo

Install ROS Bare Bones version from here.

http://wiki.ros.org/indigo/Installation/UbuntuARM

19.6 Enable WiFi with USB

Currently I was able to make it work with this module - https://www.hardkernel.com/shop/wifi-module-3/.

It's a 2.4GHz module, so 2.4GHz needs to be enabled on the router.

Install the software needed by following commands

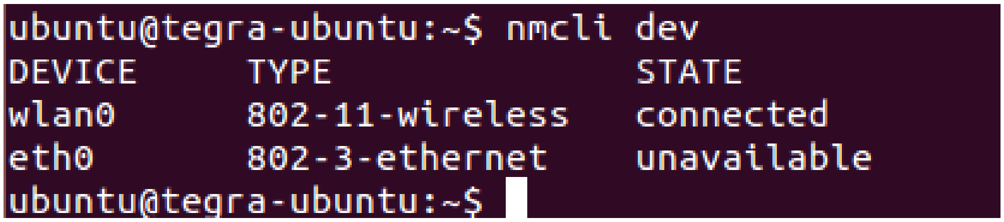
```
sudo apt-get install linux-firmware
cd
wget https://elinux.org/images/a/a8/Rtl8192cufw.bin.zip
unzip Rtl8192cufw.bin.zip
sudo cp rtl8192cufw.bin /lib/firmware/rtlwifi
```

Reboot Manifold

```
sudo reboot
```

Try this command and see if you can observe the wlan0

```
nmcli dev
```



19.7 References

1. https://dl.djicdn.com/downloads/manifold/20170918/Manifold_User_Manual_v1.2_EN.pdf
2. https://elinux.org/Jetson/Network_Adapters

19.8 Contributors

Contributor is Kuat Telegenov.

CHAPTER 20

DJI Guidance ROS setup

http://zadig.akeo.ie/

CHAPTER 21

Setting Up a ROS network: WiFi + Ethernet

21.1 Introduction

In some applications, it might be necessary to have, not one, but multiple computers connected via an Ethernet cable, and then the bunch of them connected to a remote WiFi router. This situation arose during ERL 2019 competition. I wanted to have two computers on the drone connected via an Ethernet cable, and I wanted to be able to communicate with the onboard computers via WiFi.

Why did I need two computers, you ask? One computer, the Jetson TX2, had good performance in object detection tasks because it had a GPU, and GPUs greatly enhance the speed of neural networks which are used in object detection. The other computer, the Intel NUC, had good integration with the RealSense camera which is the de facto officially supported camera for the PX4 obstacle avoidance ROS package. The idea is to have the best of both worlds by having them both on the drone. This requires setting up a network. Note that these two computers are on the drone, so it makes sense to establish a wired network between them to ensure stability of the connection, and this is why an Ethernet cable is used.

In order to communicate with the onboard computers from the ground control station (laptop), a WiFi router is used. It is desirable to be able to ping any of the devices from both networks to any other device from either network. It is also required to be able to view ROS topics on the master (in my case, the Intel NUC) from the other PCs (Jetson TX2, and ground control station).

Establishing the network described above seems like a straightforward task. However, it is not as easy as it seems. After a lot of trial and error, I am presenting one possible way of establishing this network. This may not be the best approach, but this is what worked out for me. One more note: the networking procedure described here is not specific to the hardware used. The same procedure can be followed on other hardware to achieve the desired functionality.

21.2 Hardware used

- Jetson TX2
- Intel NUC
- Laptop

- Ethernet cable

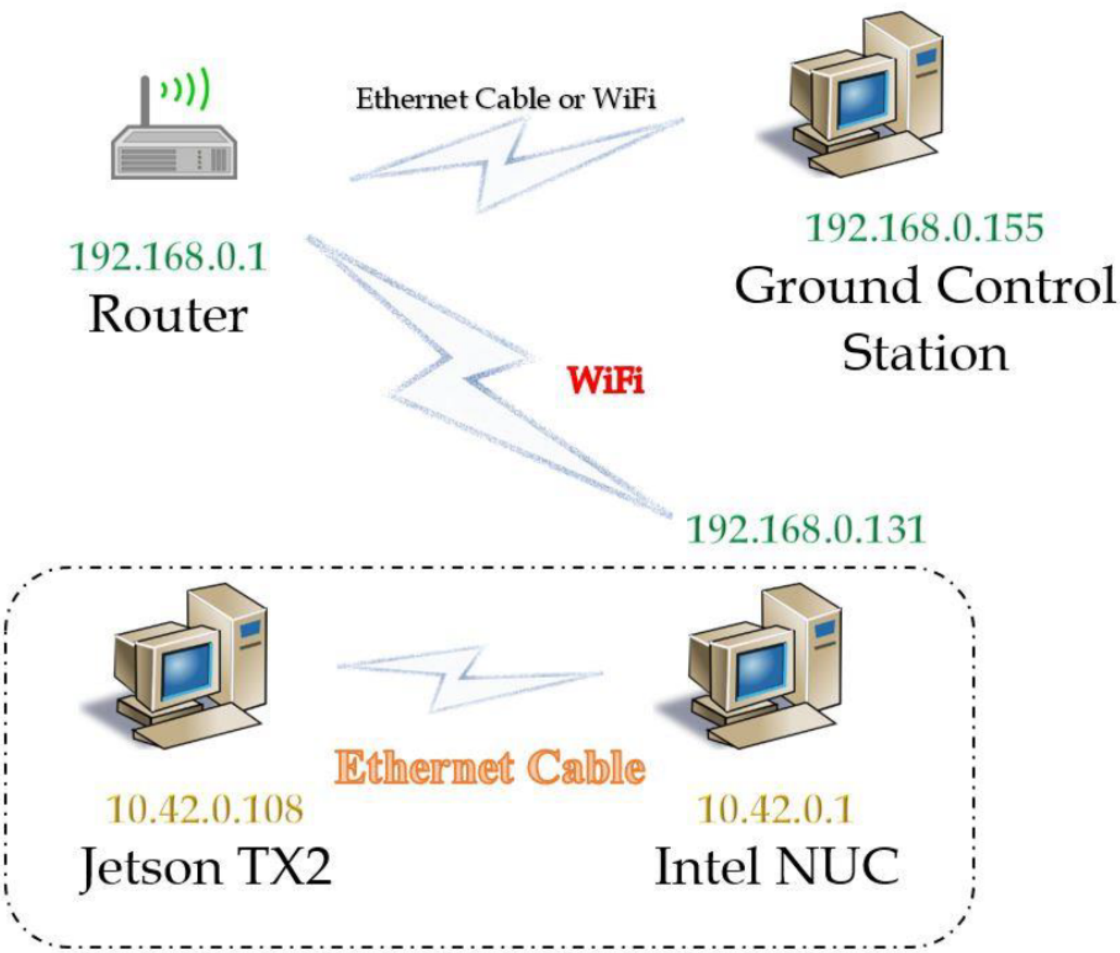
21.3 Software used

Ubuntu 16.04 on all devices

21.4 Desired functionality

- Ping any of the devices from both networks to any other device from either network
- View ROS topics on the master from all other PCs

21.5 Network Diagram

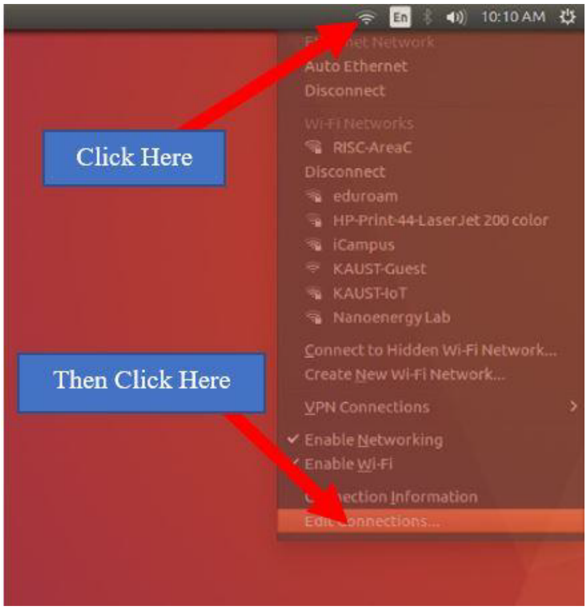


21.6 Setting the network

Connect Intel NUC and Jetson TX2 with an Ethernet cable. Make sure both PCs are on. Disable WiFi on Jetson TX2. Make sure Intel NUC is connected to the WiFi router. Try to ping Jetson TX2 from Intel NUC or vice versa. It fails. To make it work, follow the steps below:

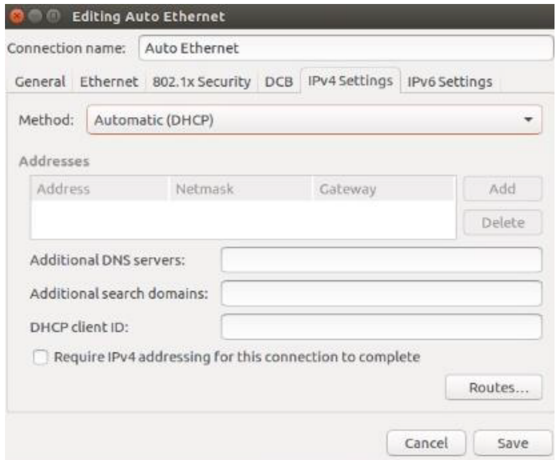
21.6.1 On Intel NUC

- Click on Networking Symbol, and click on Edit Connections:

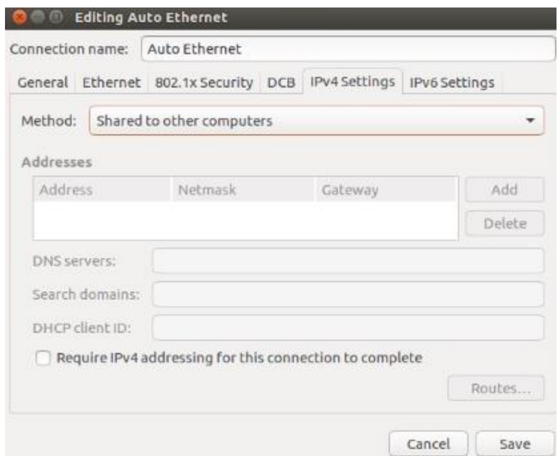


- After clicking Edit Connections, the following screen appears. Select the Ethernet connection, and click Edit.

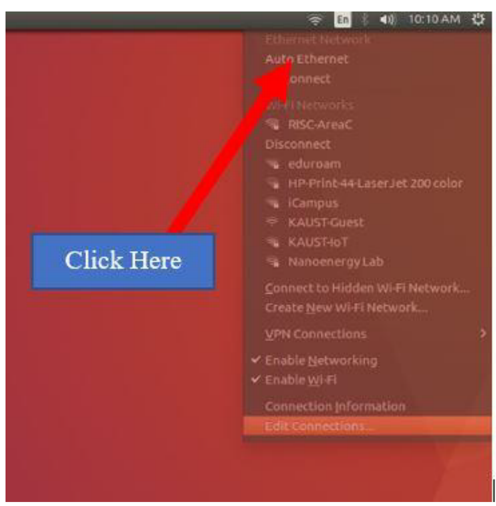
- After clicking Edit, a window appears. Go to the IPv4 Settings Tab (shown in picture below)



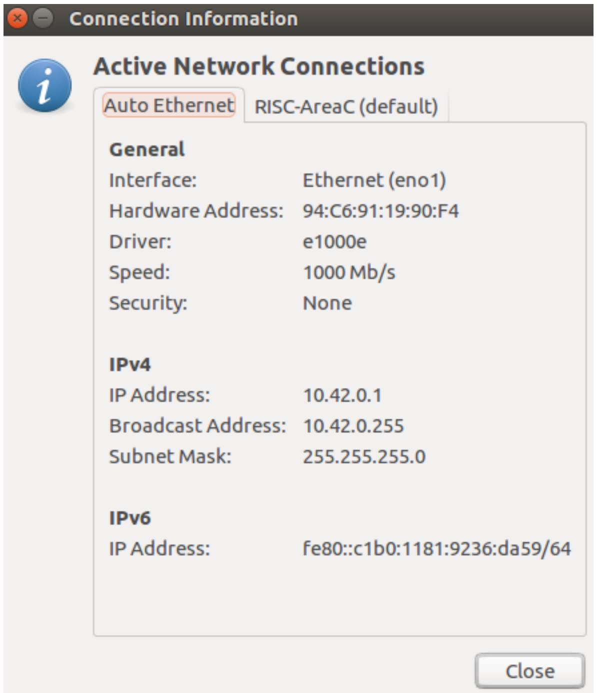
- Change Method from Automatic (DHCP) to Shared to other computers as shown below:



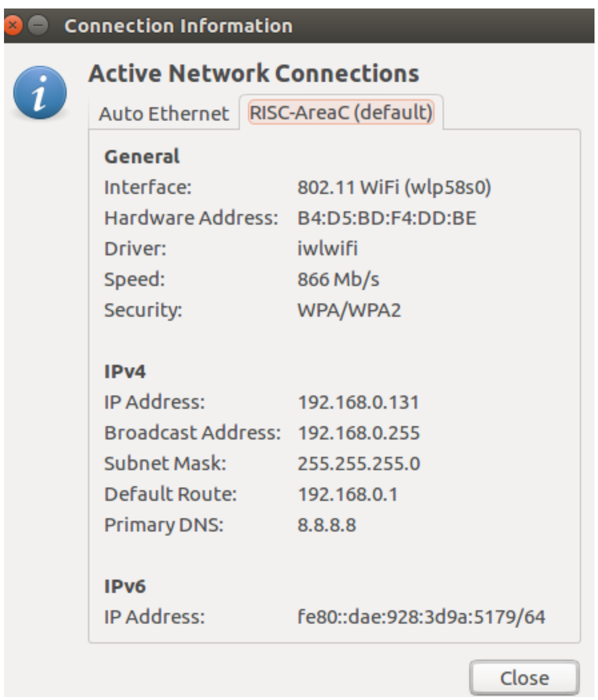
- Click Save.
- For the change in step 5 to take effect, click on Auto Ethernet (or whatever it is you called your ethernet connection) from the drop-down menu shown below:



- To verify that you have followed the steps above correctly, show Connection Information (by clicking on it from the same drop-down menu shown in step 6). You should see the following. The IPv4 address is automatically set to 10.42.0.1.



- I assume here that you have already configured your WiFi connection. Make sure you are connected to the WiFi router. In my case, this is the WiFi connection information:

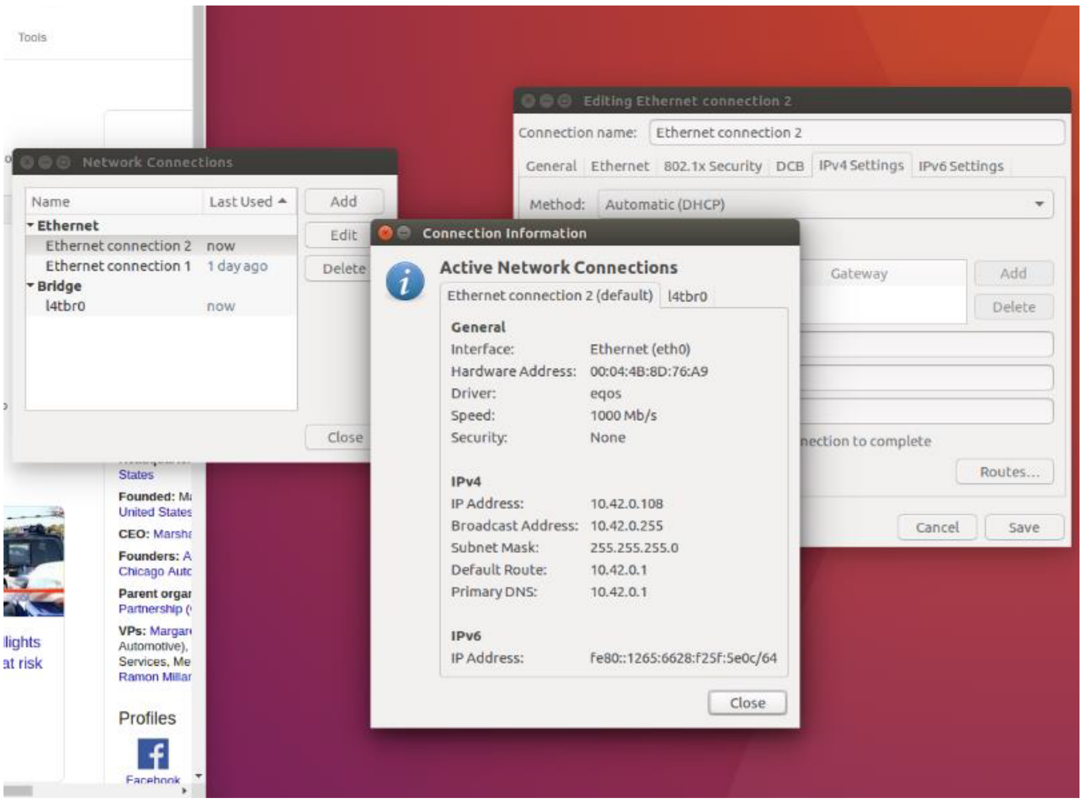


- Compare the screenshots in steps 7 and 8 with the networking diagram shown earlier to gain a better understanding of what is going on.
- Enable IPv4 forwarding
  - in `/etc/sysctl.conf`, uncomment (or add) *net.ipv4.ip_forward=1*

- or run following command in terminal `sudo sysctl -w net.ipv4.ip_forward=1`
- It's a good idea to open `/etc/sysctl.conf` in a text editor of your choice to verify the changes were applied.

21.6.2 On Jetson TX2

Following similar steps as described for Intel NUC, configure the Ethernet connection IPv4 to Automatic (DHCP). Verify that you have succeeded by pinging the Jetson TX2 from the Intel NUC, and the other way around. Also, if your WiFi router is connected to the Internet, then you will also be able to access the internet from Jetson TX2. See screenshot below:



21.6.3 On Ground Control Station

- Add static route by executing following command

```
sudo route add -net <inster_ip_of_wired_network> netmask 255.255.255.0 gw <inster_ip_
↪of_wireless_port_of_Intel-NUC_or_similar>
```

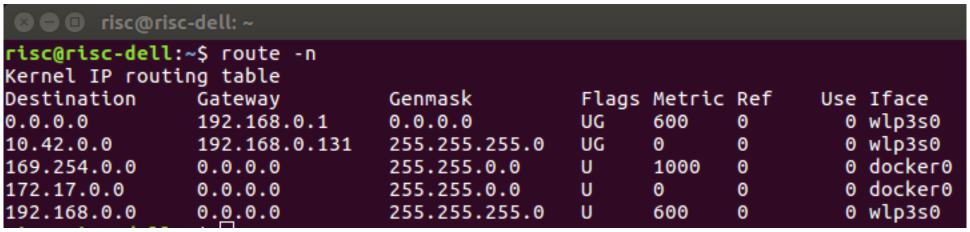
- In my case, I run the following command:

```
sudo route add -net 10.42.0.0 netmask 255.255.255.0 gw 192.168.0.131
```

- Ping Ground Control Station from Jetson TX2 and Intel NUC, and vice versa to verify things are working

Note: Adding a static route is not a permanent change. You will have to do this every time you reboot.

The figure below shows the routing table on the Ground Control Station. Check out the second entry. This was added by following step 1. The command `route -n` can be used to verify you have added the static route correctly.



21.7 Useful Networking Commands

```
ifconfig #shows network interfaces on device
nmap -sP 192.168.0.0/24 #shows all other devices on the network 192.168.0.0/24
nmap -sn 192.168.0.0/24 #same as above but does not do a port scan
sudo apt-get install nmap #installs network scanner nmap
arp -a #if you don't want to install nmap, but from my experience arp -a does not␣
↪always show all devices. On the other hand, nmap usually does.
```

21.8 ROS Communications

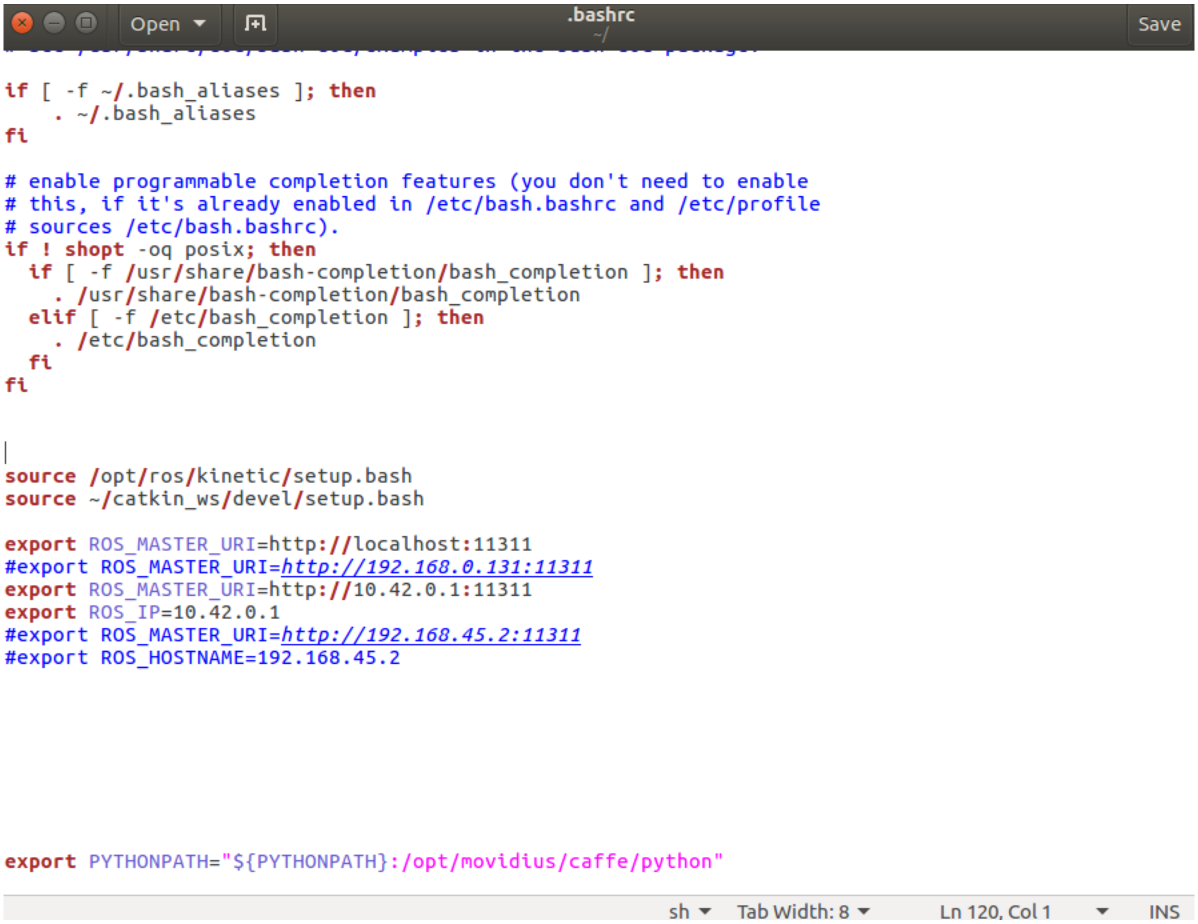
This section aims to establish ROS communication across both networks (Wired and WiFi) so that PCs on both networks can view ROS Master topics

21.8.1 Intel NUC (ROS Master)

- Add the following exports to wherever you do your exports (in my case, they are in the `~/.bashrc` file

`export ROS_MASTER_URI=http://10.42.0.1:11311`

`export ROS_IP=10.42.0.1`




```
if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi

# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if ! shopt -oq posix; then
  if [ -f /usr/share/bash-completion/bash_completion ]; then
    . /usr/share/bash-completion/bash_completion
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi

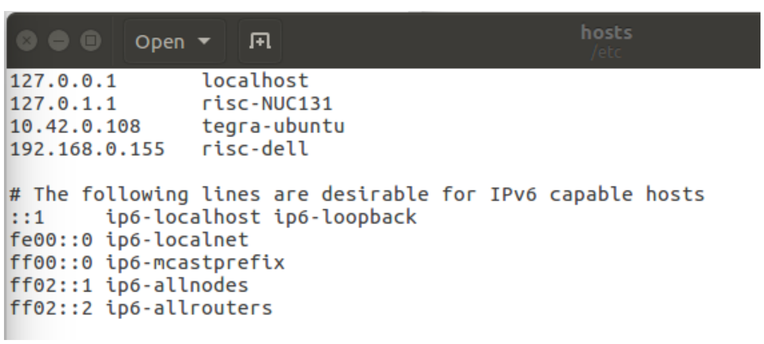

source /opt/ros/kinetic/setup.bash
source ~/catkin_ws/devel/setup.bash

export ROS_MASTER_URI=http://localhost:11311
#export ROS_MASTER_URI=http://192.168.0.131:11311
export ROS_MASTER_URI=http://10.42.0.1:11311
export ROS_IP=10.42.0.1
#export ROS_MASTER_URI=http://192.168.45.2:11311
#export ROS_HOSTNAME=192.168.45.2




export PYTHONPATH="${PYTHONPATH}:/opt/movidius/caffe/python"
```

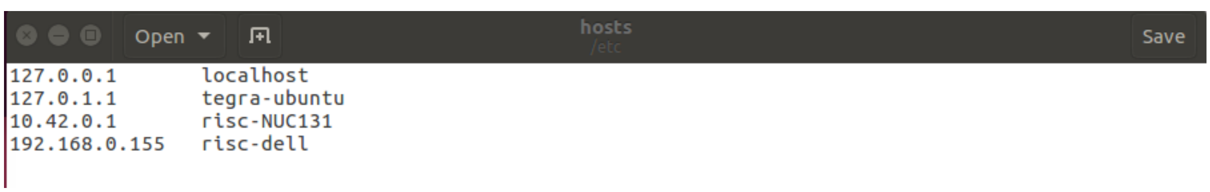

- Edit `/etc/hosts` with the hostnames and IP addresses of other devices. In my case, I have two other devices: Jetson TX2 with hostname `tegra-ubuntu` and Ground Control Station with hostname `risc-dell`. The `/etc/hosts` looks like the following:



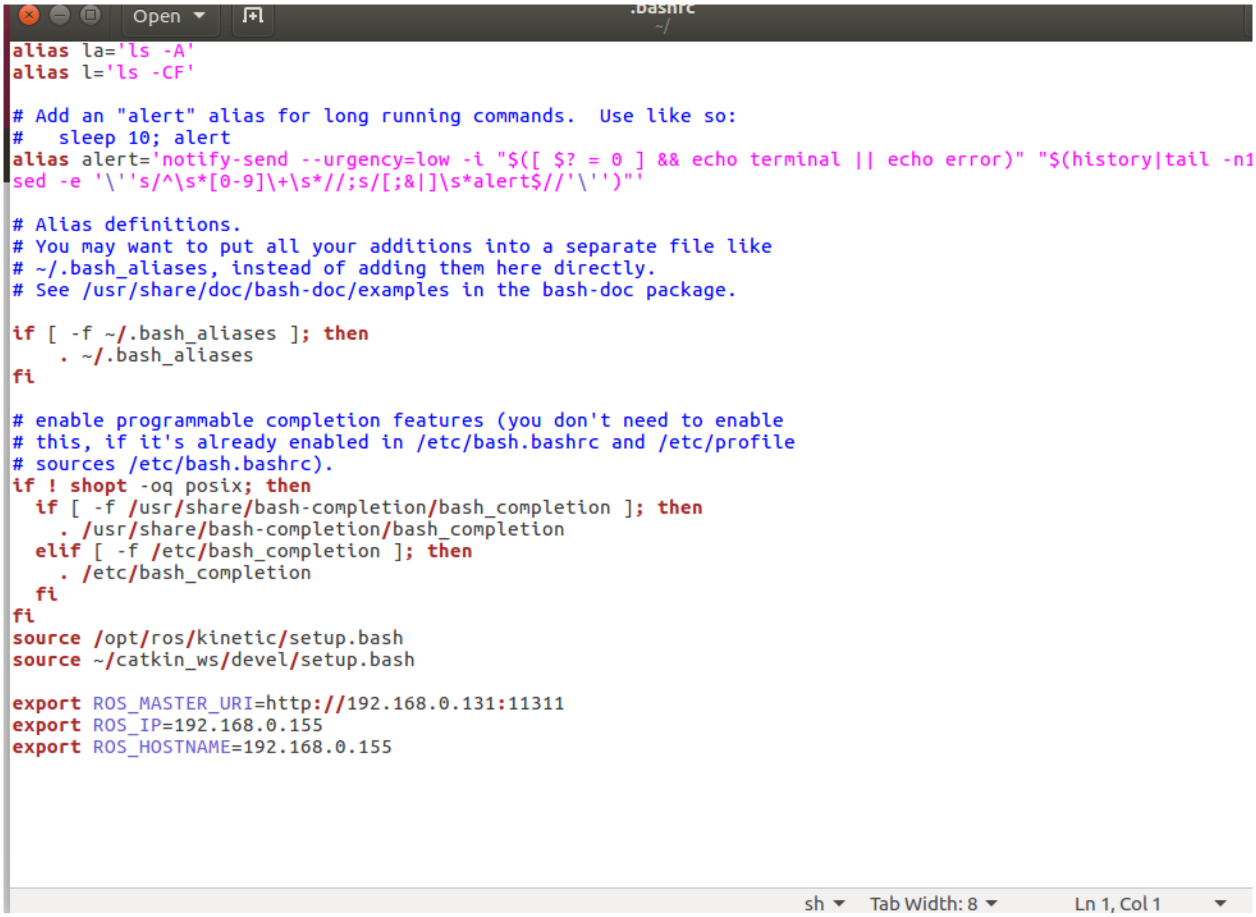

```
127.0.0.1       localhost
127.0.1.1       risc-NUC131
10.42.0.108     tegra-ubuntu
192.168.0.155   risc-dell

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

21.8.2 Jetson TX2



21.8.3 Ground Control Station



```
alias la='ls -A'
alias l='ls -CF'

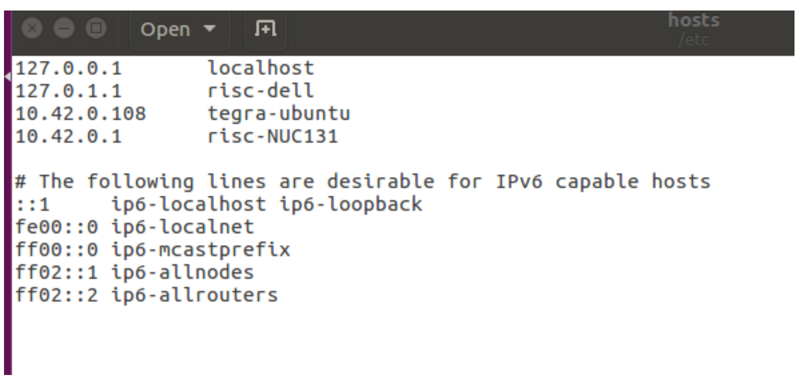
# Add an "alert" alias for long running commands.  Use like so:
#   sleep 10; alert
alias alert='notify-send --urgency=low -i "$([ $? = 0 ] && echo terminal || echo error)" "$(history|tail -n1
sed -e '\''s/^\s*[0-9]\+\s*//;s/[;&|]\s*alert$//'\'')"'

# Alias definitions.
# You may want to put all your additions into a separate file like
# ~/.bash_aliases, instead of adding them here directly.
# See /usr/share/doc/bash-doc/examples in the bash-doc package.

if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi

# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if ! shopt -oq posix; then
  if [ -f /usr/share/bash-completion/bash_completion ]; then
    . /usr/share/bash-completion/bash_completion
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi
source /opt/ros/kinetic/setup.bash
source ~/catkin_ws/devel/setup.bash

export ROS_MASTER_URI=http://192.168.0.131:11311
export ROS_IP=192.168.0.155
export ROS_HOSTNAME=192.168.0.155
```



```
127.0.0.1       localhost
127.0.1.1       risc-dell
10.42.0.108     tegra-ubuntu
10.42.0.1       risc-NUC131

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Important: Don't forget to source your `.bashrc` file after and if you edited it.

To verify that ROS communication is established, run anything on the ROS Master (Intel NUC) and see if you can list topics from other PCs. For example, run a ROS camera package, and try to view the video stream through rviz on other devices (Jetson TX2 and Ground Control Station). Note: currently, if you run a ROS package on Jetson TX2 (Not the ROS master), you will be able to view the topics from the Ground Control Station, but not their contents.

What if you want to launch a file from Ground Control Station, and you want to be able to select which machine to run this node. Look no more. The answer is here:

http://wiki.ros.org/roslaunch/XML/machine

21.9 References

None of which I strictly followed. I just got inspiration from these references in establishing the network and in writing this document. Things got working by trial and error and luck.

https://risc.readthedocs.io/2-networking.html

https://github.com/ethz-asl/mav_dji_ros_interface/wiki/NVIDIA-Jetson-TX2-integration

Some Good Textbooks about Networking (first book = linux networking, second book = networking in general)

http://linux-training.be/linuxnet.pdf

http://iips.icci.edu.iq/images/exam/Computer-Networks—A-Tanenbaum—5th-edition.pdf

21.10 Contributors

Main contributor is Tarek H. Mahmoud.

CHAPTER 22

Pick and drop demo

Pick and Drop is a demonstration of object transportation using a drone in an indoor setup. The control can be either via a human pilot who controls the drone using a joystick to fly the drone, pick, transport and drop the object, or fully autonomous using vision feedback.

The following setup is assumed.

- Indoor localization system (optitrack)
- A drone that is equipped with a PX4 autopilot and an arduino-controlled customized gripper.
- An object that is magnetic (can be picked by a permanent magnet)
- A ROS-compatible joystick for manual control
- A ROS-compatible camera for vision feedback, for autonomous mission
- ROS Kinetic, Ubuntu 16 on ODROID XU4, or a similar onboard computer

22.1 Dependencies

- vprn_client_ros
- apriltag2_ros
- cv_bridge
- rosserial

22.2 Installation

- Make sure you install the required dependencies above
- Clone this package into your ~/catkin_ws/src and build it
- The arduino code that controls the gripper is in the gripper_joystick folder.

22.3 Experiment

- Place markers rigidly on the drone, and define a rigid body in Motive
- Stream the rigid body info using VRPN, and make sure that Up axis is the z-axis
- It is assumed that you have an onboard computer which runs mavros, which can be used to feed the rigidbody pose from motion capture information to PX4.

Note: Always double check that you can hover the drone in **POSITION** flight mode, before you execute the experiments in **OFFBOARD** mode.

22.4 Manual control (Drone 1)

- Make sure that you give the joystick permissions (we used Logitech F710).

```
sudo chmod a+rw /dev/input/js0 # Check the input device number
```

- The right analog stick is for x/y (position) motion. The left stick is for height. The red button is for disarm. The green button is for autoland. The down button on D-Pad is for dropping, if the object is picked (detected by the button on the gripper).

- Run the following command on the onboard computer. Double check the addresses for joystick, arduino, FCU, Ground Control Station, and name of the Rigid Body from Mocap system.

```
roslaunch pick_drop_demo start_manual_test.launch
```

22.5 Autonomous mission (Drone 2)

- Run the following command on the onboard computer. That will start the autonomous mission with takeoff action to 1m height. Double check the addresses for arduino, FCU, Ground Control Station, and name of the Rigid Body from Mocap system.

```
roslaunch pick_drop_demo start_autonomous_mission.launch
```

22.6 Contributors

Main contributors are Asmaa AlSaggaf and Mohamed Abdelkader.

CHAPTER 23

Taranis + DJI Naza/N3

23.1 Introduction

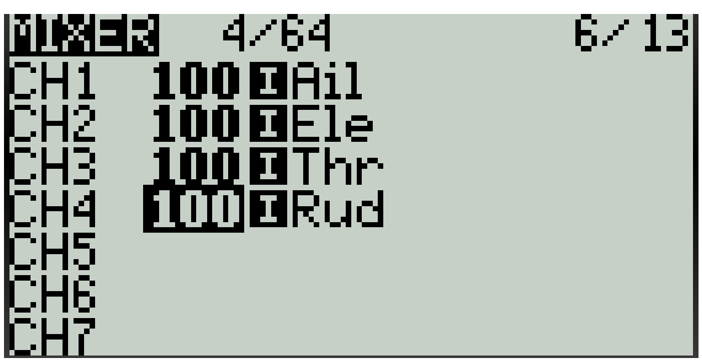
This tutorial explains how to setup Taranis transmitter with DJI Naza/N3 flight controllers. The setup was tested with Taranis X7Q, DJI Naza-M v2, and FrSky D4R-II receiver. Also, should work with any FrSky receiver, Taranis transmitter, and DJI Naza series flight controllers.

23.2 Binding process

Please follow the official instructions from FrSky website to bind receiver with transmitter. On the transmitter, set **Internal RF** mode to **D8** if using D4R-II and **D16** if using X8R.

23.3 Setup

In the menu of the Taranix X7Q transmitter go the **MIXER** page and change it to match following picture.



Assuming the FrSky receiver is connected to the X2 port of the flight controller.

23.4 Contributors

Main contributor is Kuat Telegenov

CHAPTER 24

Appendix: RISC AUV System Manual

Details of RISC AUV system is discussed here, including the structure of simulation system, hardware system, localization systems, all ROS software packages with illustration, the network structure of the system and everything related.

24.1 ROS/Gazebo simulation system manual

Main functions and operating manual of simulation system is in this section.

24.1.1 MultiROV

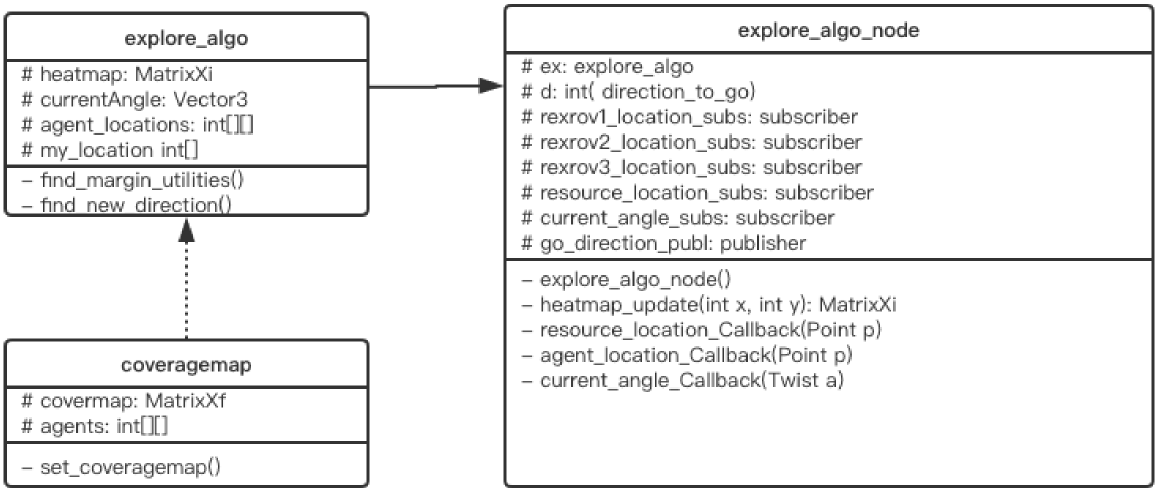
MultiROV contains game-theoretical algorithm, protocols to communicate with ROS environment, and other essential software components. It is created based on the principle of separating algorithm parts and ROS parts so that each part can be modified individually. Note that there are two branches in this repository. The master branch is for the simulator, and the BlueROV branch is for the hardware test. However, BlueROV branch is written more cleanly and abandoned many parts that are not necessary. So I recommend using BlueROV branch for both simulation and hardware. The user can refer to commit history of this repository for detailed development process as I have clear notes while creating this project for future reference.

It's class diagram is shown as below.

[class_simulation]

Note that only important members and methods of the classes are presented. Here are three main components:

- explore_algo class is the high-level algorithm class, deciding for an agent with local information of resources and other agents. It builds a heatmap with local resources and a coveragemap(not working as a class member but a local variable inside a method) with visible local agents. Last two steps of its computation are listed in its methods in the class diagram
- coveragemap class builds the nearby coverage status for an agent with local information of other agents. Methods include
  - set_coveragemap(): based on nearby agents to compute coverage status



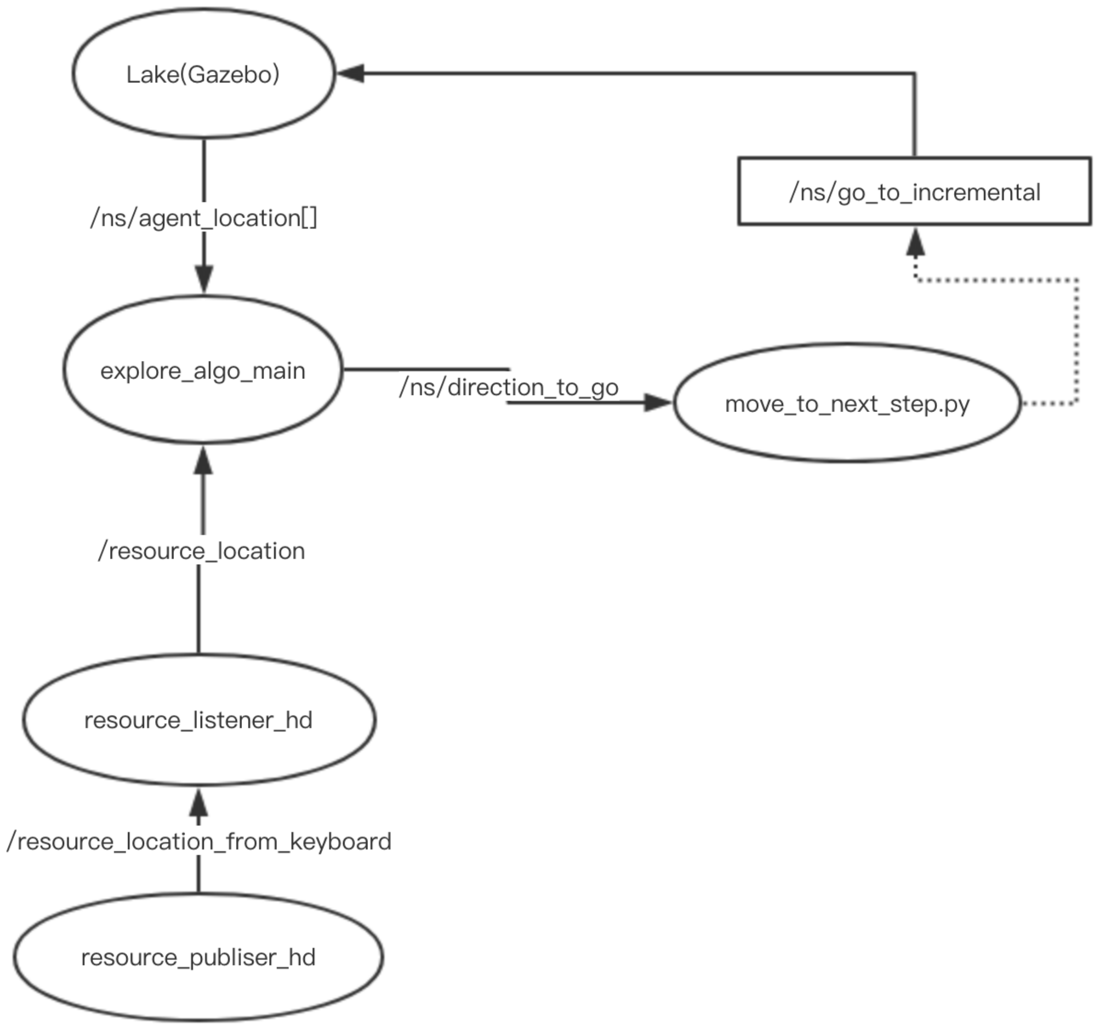
- explore_algo_node class has explore_algo as its member to high level algorithmic computations. Other subscribers are used to subscribe resource locations, agent locations and current direction from Gazebo topics. Publisher is used to publish computed command to a controller node to send incremental control service for simulation and controller node for hardware case. Important methods include:
  - heatmap_update(): function needed for resource_location_Callback(), update local heatmap of member ex with locations of sensed resources. Note static resource hardcoded at (3,4) here but still keeps secret for the agent when it's out of the sensory range.
  - resource_location_Callback(): triggered once resource location is received from Gazebo/other outside nodes. Update heatmap in ex.
  - agent_location_Callback(): triggered once agent location is received from Gazebo/other outside nodes. Will execute algorithm and send command the controller, either in simulation or in hardware.
  - current_angle_Callback(): triggered once current is changed and related topic is publishing. Will modify a parameter in ex to change the utility computation.

One thing to notice is the way we express commanded directions in this setting. We use
2 5 8
1 4 7
0 3 6
i.e., 4 means stay, 7 means go one step in the x-direction, etc.

24.1.2 UUV simulator

UUV simulator as mentioned before is an open source software package to simulate underwater robots and its working environments. As it's a big complicated package, we will only discuss some issues when using its related parts. Notice this package is modified by me so please use the version at https://github.com/luym11/uuv_simulator as it's the version in RISC marine workstation. Also, there are some pre-requirement software packages for installing this package, they are installed correctly on RISC marine workstation. There will be some instructions about this in the last part of this appendix.

We will structure this part with ROS nodes need to run and related explanation. Then we will draw a ROS node graph as Fig*[simulation_nodes]* to show the relation between these components.

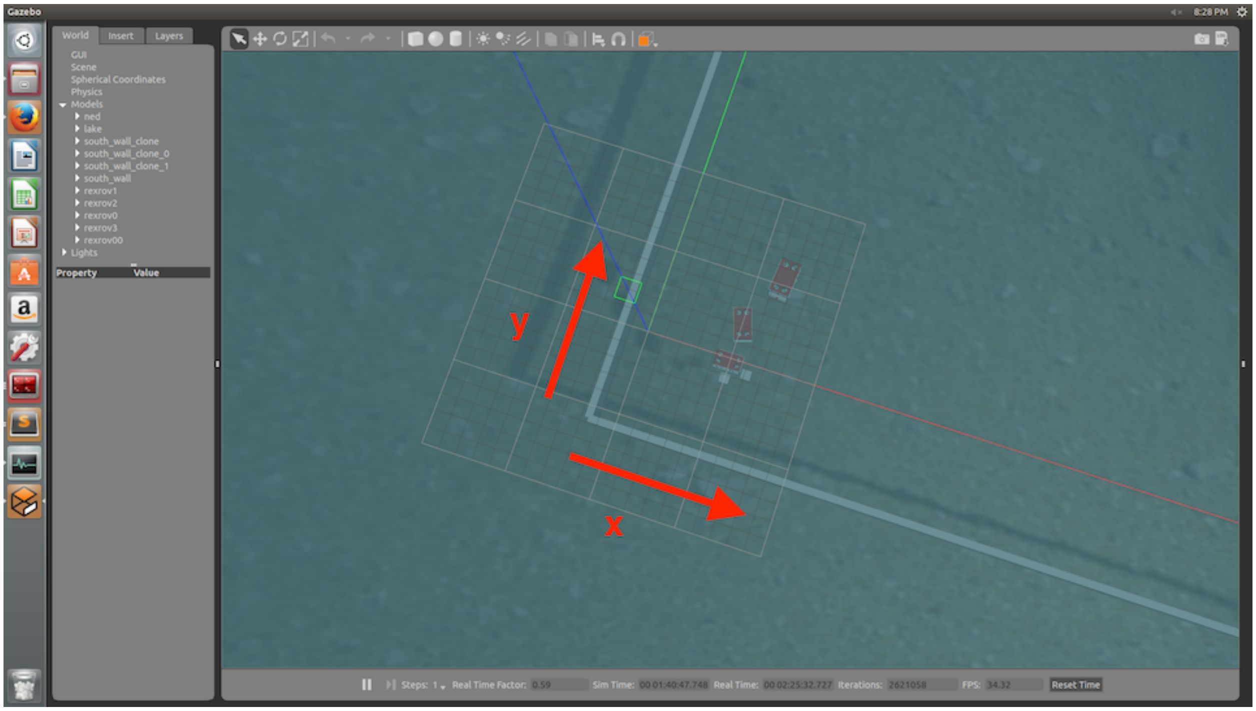


[simulation_nodes]

- roslaunch multirov lake.launch: This launch file loads Gazebo world, it's appearance and simulated time. After loading this file, Gazebo environment will be open. Note that x and y axes are already set in Gazebo, we also use walls to indicate that as *[rov_multi]*.

  [rov_multi]

- roslaunch multiagent_simulation multiagent.launch: this file loads AUV models, resource models and corresponding controllers. RViz can also be loaded from here. In details, we launched (in terms of namespace)
  - rexrov1, 2,3: spawn robot model in Gazebo; publish its states to Gazebo; publish its position to ROS topic(using agent_listener node running with it); simulated dp controller.
  - rexrov0: spawn resource model in Gazebo(originally at (5,5)); publish its states to Gazebo; joystick node to control it (notice agent_listener is not here because the code for resource was developed earlier and used another method for publishing the location).
  - rexrov00: spawn resource model in Gazebo(static at (3,4)); publish its states to Gazebo.



- rosrun multirov resource_publisher_hd: let the movable resource controlled by publishing to the topic resource_location_from_keyboard. The way to change resource location is `rostopic pub /resource_location_from_keoard geometry_msgs/Point "x: 4.0 y: 4.0 z: -30.0" -r 1` and different from the joystick that can also change the location of the movable resource, this change with a keyboard is instant.
- rosrun multirov resource_listener_hd_node: subscribe from above publised topic and republish to the topic resource_location. Our previous method was a more complicated way of implementing agent_listener node by subscribing rexrov0/base_stabilzed and republish to our own topic resource_location. Now we move to this method for compatibility because in hardware phase we can't get positions from Gazebo neither the existence of related topics. For vehicles they can be localized by our method, for virtual targets, this is the best way to write this so that it can be used both in simulation and hardware. More details can be found in commit comments in BlueROV branch.
- roslaunch multirov explore_environment.launch for three vehicles: Executes previous mentioned node explore_algo_node_main and a incremental controller which calls the service ns/go_to_incremental.

Also, this package supports useful topics and services, for example:

- Add current: rosservice call /hydrodynamics/set_current_velocity "velocity: 1.0 horizontal_angle: 1.7 vertical_angle: 0.0" and this will be published to related topics as if the ROVs have sensor to sense it.
- go_to service: command the vehicle to a specific position in Gazebo.

24.2 Hardware system manual

In this section, we will discuss the hardware implementation phase of this project. As this system consists of many parts, we will talk about them separately.

24.2.1 BlueROV

Kit Assembly and common issues

Please refer to their official website for assembly while noticing following points:

- It's recommended to test each ESC and motor before sealing the enclosure. It will be very hard to change any of them if the ROV is fully assembled.
- Fathom-X Topside board always needs to be powered by Mini USB, or it will not work.
- Organize the tether wire cleanly and don't let it twist when doing experiments, or much time will be wasted on untangling them.
- When opening the enclosure, remember to remove the penetrator first; when closing the enclosure, remember to close the penetrator after closing the cap. It's for water proof sealing purpose.
- Use 7.0Ah, 14.8V batteries in the lab as they last much longer than the others.
- Do a vacuum test every time before submerging.
- Motor direction can be reconfigured through QGroundControl software and don't need to change its wires on hardware.

Network setup and companion computer

Here we are using Fathom-X to extend the ethernet longer and communicate with the Raspberry Pi inside the BlueROV. BlueROV originally comes with a companion Raspberry Pi with a system image that only allows joystick control through QGroundControl ground station, which is not what we desire. So we reimaged the Raspberry Pi with an Ubuntu Mate system, then installed related software packages there, including ROS Kinetic and BlueROV ROS package (modified) from https://github.com/luym11/bluerov-ros-pkg.

We mainly use two parts of this package. For BlueROVs, we will launch bluerov bluerov_r1.launch locally, which loads state publisher, MavROS that talks to ArduSub firmware, imu and camera equipped on the ROV. For controller from ROS via MavROS (both joystick and codes), we launch bluerov_apps teleop_f310.launch on ROS master machine because it needs a joystick for emergency operation, change of mode, arm/disarm, etc. This modified controller node can additionally take direction_to_go as input from ROS topic and control the ROV to go towards that direction with a pre-set speed by publishing to rc_override topic as the joystick does. Note this also means we can directly publish to this topic to control the ROV from the command line.

Note that for some version of ArduSub firmware, the ROV can not take commands from MavROS. For now only ROV1 associated with IP 192.168.0.111 has the correct version of firmware. This will be checked further.

For hardware basic testing, we have a water tank in RISC lab. To use it, please use the mountain climbing rope attached to both the ROV and the beam on top of the tank in case it sinks. Normally testing operation can be done by only one person as the ROV will automatically float on the surface when disarmed.

Instead of the network configuration used in their manual which can only control one ROV at a time, network interfaces of them are reconfigured and connected to RISC marine router with pre-assigned static IP addresses. Note that we will connect all the devices through this RISC marine router with static IP address. A detailed list will be included in the last part.

Raspberry Pi OS image (software packages configured) used here is stored in RISC Google Drive, after flashing, remember to change

- .bashrc for ROS_IP and ROS_MASTER
- interfaces in etc folder for IP address
- bluerov1.launch for ground station IP and target number which is used in accessing multiple ROVs from QGroundControl

24.2.2 Localization system

As mentioned before, a localization system is essential for both knowing the positions of agents and resources. Also it's needed for waypoint feedback control of the ROVs. We will introduce two methods we have so far.

Tritech USBL

For USBL method, we use Tritech USBL devices. Transponders will be installed on ROV as shown in Fig*[serial]* and powered from the battery there. Transceiver is powered by it base controlled by software on windows machine and data will be transferred to ROS master PC from serial port. Related ROS package is at https://github.com/luym11/RISCusbl. So the overall architecture is shown in Fig*[usbl_archi]*.

[usbl_archi]

When using this system, please use the specifically made serial port reader as Fig*[usbl_on_rov]* for its voltage level.

[usbl_on_rov]

Vision-based system

As the defects of USBL system mentioned before, we finally used a vision-based method for this stage of hardware test. Here we chose to use Apriltags to mark the ROVs and use a fisheye camera with related packages to give relative locations of each marker. Then we use a ROS node called location_bridge to publish these locations to agent_locations[ ] topics as we did for Gazebo, thus close the control loop.

First, we need to choose a proper camera and calibrate it. After testing different kinds of camera, we finally chose the fisheye camera and calibrated it using a ROS camera calibration package. This localization system is installed on a DJI matrice 100, with an on-board computer as shown in Fig*[dji_top]* and Fig*[dji_down]*.

[dji_top]

[dji_down]

Then package at https://github.com/luym11/apriltags2_ros is used to detect markers. Test indoor and outdoor showed its good performance as shown in Fig*[marker_out]*.

[marker_out]

We used Odroid with WiFi communication to RISC marine router to send detected locations to ROS master computer. Three software components are running on the odroid:

- The USB camera node to publish camera image camera
- image_proc package to do image rectification
- Detection code that gives relative location of each marker to the center of the camera

The odroid image is also stored in RISC Google Drive.

The software running on PC is a location_bridge node, remap these coordinates and publish them to agent_locations[ ] topics instead of the Gazebo environment. With this architecture, we can create a closed control loop.

The overall system architecture is shown in Fig*[ros_hard]*

[ros_hard]

All the commands need to run for one robot open-loop test with this set up are as follows, note the algorithm part is not included in the test now, but as we have the localization system, there is not too much work to close the loop as the architecture graph shows.

- On ROS master machine

```
roscore
roslaunch bluerov_apps teleop_f310.launch
rosrun image_view image_view image:=/tag_detections_image: to
monitor the view of the camera
```

- On Odroid

```
roslaunch apriltags2_ros rov.launch
```

- On BlueROV

```
roslaunch bluerov bluerov_r1.launch
```

So the network architecture of this system is Fig*[network]*

24.3 Others

24.3.1 Data recording and representation

It's recommended to use rosbag and rqt_multiplot to record and represent data, respectively.

24.3.2 list of software packages and OS images

Software packages

A list of all software packages used (with hyperlinks). They are all host on my account publically on Github. Will be forked to RISC account.

- MultiROV
- UUV simulator (modified)
- BlueROV packages (modified)
- Apriltags detection package
- USBL serial reader

OS images used

- Original OS image for BlueROV (just for archive purpose)
- Ubuntu 16 Mate with ROS, MavROS and BlueROV package: for Raspberry Pi
- Ubuntu 16 Mate with ROS and AprilTag package: for Odroid

24.3.3 Carrying list for outdoor test

As there will always be something forgotten, a list of carryings when going outdoor test is created and maintained.

- School bus key
- DJI Matrice 100, 2 batteries, RC, connection wire with the smartphone, attached Odroid (with WiFi stick and batteries) and camera, attached camera
- Odroid backup: with WiFi, power cable, a camera with USB cable
- Odroid console cable
- SD card reader
- Tapes
- Battery checker
- Ethernet cables
- ruler
- zip ties
- RISC marine router with battery and power cable
- Apriltag markers
- Linux PC (RISC marine laptop)
- ROVs with tether, Fathom-X power cable, ethernet cable, batteries
- Logitech joystick

24.3.4 Equipment list and backups

- Linux ROS Master risc@192.168.0.195, risc
- ROV1 risc@192.168.0.111, risc; gcs target 1
- ROV2 risc@192.168.0.112, risc; gcs target 2
- ROV3 risc@192.168.0.113, risc; gcs target 3
- ROV2 Test Pi with a ArduSub installed Pixhawk risc@192.168.0.112, risc; gcs target 2
- Camera Odroid odroid@192.168.0.190, odroid
- Camera Odroid backup odroid@192.168.0.180, odroid

24.3.5 UUV dependencies troubleshoot

Look at the log, reinstall essential packages, modify CMakeLists. Remember to source the bashrc everytime redo catkin build to make changes really effect.

Eigen 3 issues

Can't find related CMakeLists
Change related CMakeLists as
-find_package(Eigen3 REQUIRED)

+find_package(PkgConfig)
+pkg_search_module(Eigen3 REQUIRED eigen3)
Can't find eigen/core

- Make a new soft link to src
- modify include_directories(include $catkin_INCLUDE_DIRS$Eigen_INCLUDE_DIRS)

Other dependencies

teleop issue
Rebuild this package from source or use apt-get

24.3.6 Others

- Some version of firmware doesn't allow offboard mode. In this situation, if the vehicle still operates with RC commands, it's mostly through QGroundControl. Notice key settings in these two situations are different.

CHAPTER 25

How to start bluerov simulator

To setup the environment for Gazebo simulation and BlueROV2 using SITL and Ardusub follow the following instructions.

2- run from ardupilot ./was build

3- install mavproxy and mavlink if required https://gist.github.com/monabf/bc04b7ab366f812c645bf0aa6f22c8de

4- run from ArduSub directory:

```
../Tools/autotest/sim_vehicle.py -f gazebo-bluerov2 -I 0 -j4 -D -L RATBeach --console
```

5- run from bluerov_ros_playground:

```
source gazebo.sh
gazebo --verbose worlds/underwater.world -u
```

6- add apm.launch to any package you have in ros or create a new package for that. Launch file available here.

```
source devel/setup.bash
export LD_LIBRARY_PATH=/opt/ros/kinetic/lib
roslaunch rplidar_ros apm.launch
```

7-run:

```
rosrun mavros checkid
rosrun mavros mavparam set SYSID_MYGCS 1
```

8-run:

```
rosrun mavros mavsafety arm
rosrun mavros mavsys mode -c MANUAL
```

by Sarah Toonsi

CHAPTER 26

Video Cameras

26.1 Recording the video

- Right click with the mouse on the live view screen to open the `Menu Bar`.
- Enter the password as prompted.
- Choose `Manual record`, that will initiate manual recording. Choose the cameras you want to record and press `Apply`.

26.2 Saving video files to external USB

There's Sandisk 128GB external USB drive plugged into the NVR.

- Right click with the mouse on the live view screen to open the `Menu Bar`.
- Choose Menu (Home icon), that will open the main menu.
- Enter the password as prompted.
- Select the `Search:Backup` tab.
- Choose the camera(s) you want to copy footage from.
- Set your `Start Time` and `End Time`.
- Select `Backup`.

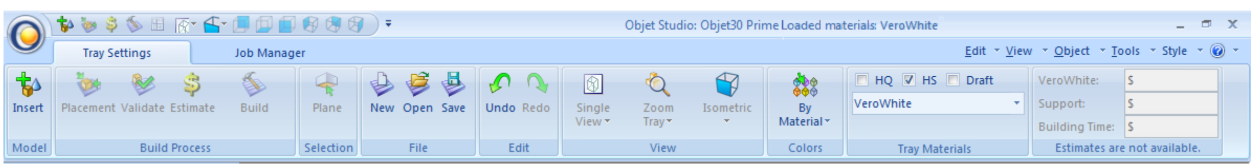
The backup file list will show you a list of all the video events between the start and end times you've selected. All the ticked files will be the part of copying process. Press `Next` and then `Start`, that will start copying selected video files to USB drive.

CHAPTER 27

3D Printing

27.1 Objet30 Prime

- Open Objet Studio application from Desktop
- Click **Insert** from Tray Settings tab to place an object on the build tray. In Insert dialog box, choose your STL file and units of your file



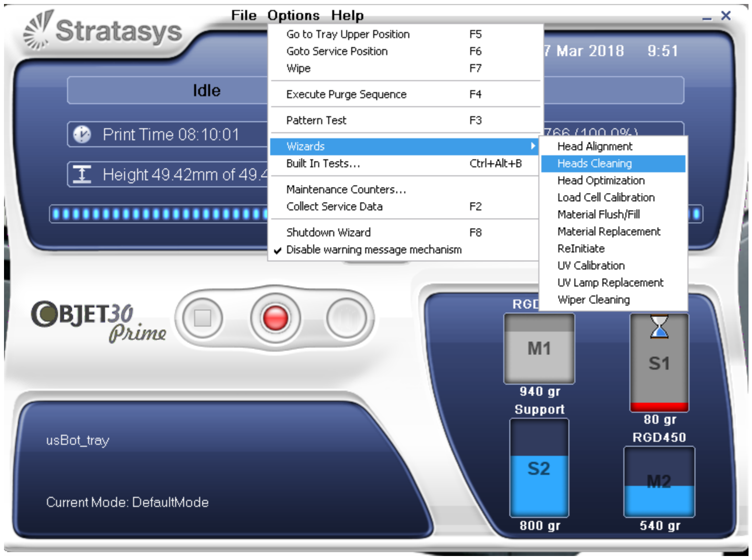
- Click **Placement** to automatically arrange objects on the build tray
- Click **Validate** to check that the tray is "valid" and can be printed.
- Click **Estimate** to calculate the time and material resources needed for producing trays before sending them to the printer
- Click **Build** to start printing

27.1.1 After each printing

Cleaning the print heads and the roller

To maintain the Objet30 printer in optimum condition, clean the print heads after every print job, when you remove the model from the build tray.

- Start the **Head Cleaning** Wizard from the Options menu



- Follow the instructions on the wizard screens, and select the confirmation check boxes, click Next
- When the screen with **Clean Printer Components** appears, open the cover

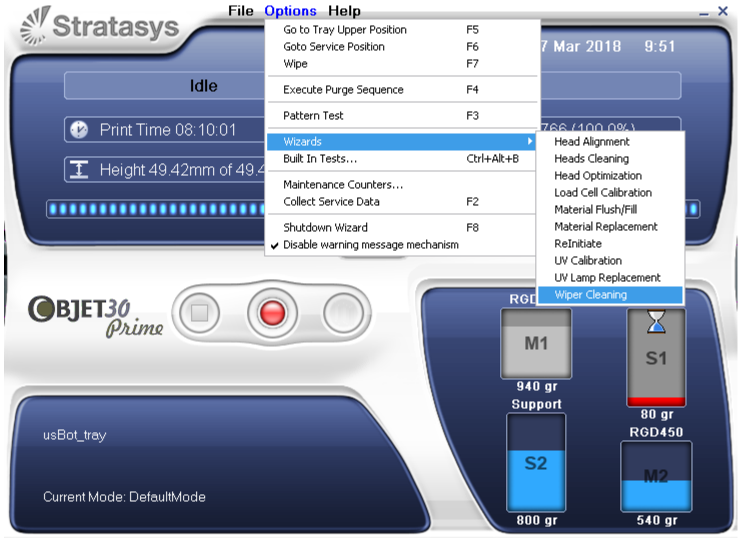
Warning: The print head orifice plates (bottom surface) may be hot. Do not touch them with your bare hands, and proceed with caution.

- Place the mirror on the build tray
- Put on the gloves
- Soak the cleaning cloth with alcohol
- Clean the orifice plates, with a back-and-forth motion. Use the mirror to make sure that you have removed all of the residue material
- Clean the entire roller surface, by rotating it as you clean
- When you have finished cleaning, select the confirmation check boxes in the wizard screen and click Next
- Remove the cleaning materials and mirror from the printer and close the cover
- Select the confirmation check boxes in the wizard screen and click Next. The head-purge cycle begins. When this is complete, the final wizard screen appears
- Click Done to close the wizard

Cleaning the wiper

A rubber wiper removes excess material from the print heads after the purge sequence. This is done automatically before each print job. You should clean the wiper and surrounding area.

- Start the **Wiper Cleaning** wizard from the Options menu.



- Make sure that the build tray is empty, and close the printer cover. Confirm this in the wizard screen and click Next
- When the next screen appears, open the cover
- Put on the cleaning gloves
- Using a generous amount of alcohol on the cleaning cloth, remove any material remaining on the wiper and the surrounding area
- In the wizard screen, confirm that the wiper blade is clean, and click Next
- Remove all tools and cleaning materials from the printer, and close the cover. Confirm this in the wizard screen, and click Next
- Click Done to close the wizard

Note: The routine maintenance tasks are performed only by lab engineer. You may refer to Objet30 user guide document for more detailed instructions.

27.2 Ultimaker3 Extended

How to Print

- Install Cura software (Windows, Linux and OSX are supported) or use it on the iMac in Area A.
- Save your 3D model as a STL file from your Computer-aided design (CAD) software.
- Open STL(s) files in the Cura software.

Note: The routine maintenance tasks are performed only by lab engineer. You may refer to Ultimaker3 user guide document for more detailed instructions.

CHAPTER 28

CNC Machine

Under maintenance.

CHAPTER 29

Drill Press

CHAPTER 30

Dremmel

CHAPTER 31

Circular Saw

Do not use for now.

CHAPTER 32

References

The following are some links to materials that can be useful.

32.1 VIO: Visual Intertial Odometry

- ROVIO
- https://github.com/ethz-asl/mav_dji_ros_interface
- SVO

32.2 SLAM: Simultaneous Localizatoin and Mapping

- ORB-SLAM
- LSD-SLAM
- RTABMAP

32.3 Obstacle Avoidance

- PX4

32.4 Other Vision/AI projects

- NVIDIA Redtail