Aeromodelling Club IITK

Drone Bootcamp

Project Mentors:

1.Mohit Anand
2.Moin Ahmed
3.Pranshu Singhal
4.Raj Agarwal
5.Prathamesh Thakare
6.Ashwin

Team Members:

1.Shreenaga Tejas
2.Mayank Saini
3.Pranshu Gaur
4.Aditya Anand
5.Utkarsh Aditya
6.Nitin vedwal
7. Varsha Rani
8.Faheem Nizar
9. Devansh Agarwal
10. Gargi naladkar
11.Sweta Kumari
12 Ayush Dhingra

13. Pranay Kansal
14.Palmate Aditya
15.Akansha Yadav
16. Saurabh Kumar
17.sheeshram choudhary
18. Aditi Sinha
19.Prabuddha Singh

QUADCOPTER

The term “drone” usually refers to any unpiloted aircraft. Sometimes referred to as “Unmanned Aerial Vehicles" (UAVs), these crafts can carry out an impressive range of tasks, ranging from military operations to package delivery. Drones can be as large as an aircraft or as small as the palm of your hand.A quadcopter or quadrotor is a type of helicopter with four rotors.

Basic avionics of drone:-
➢BLDC motor

➢Electronic Speed Controller

➢Receiver

➢Transmitter

➢Battery

➢IMU

➢Flight controller

➢Radio telemetry

Controlling basics in a drone:

- Accelerometer: used to determine position and speed
- Barometer:Used to determine magnetic field
- Magnetometer:Altitude is determined from pressure
- gyroscope:Detects angular velocity

Motion of a drone can be determined by DOF(Degree of Freedom) pitch, yaw, roll, thrust, which are basically controlled by varying rotor speeds.

❏ Roll- Rotation about front to back axis

❏ Pitch- Rotation about right to left axis

❏ Yaw- Rotation about vertical axis

The basic principle to move a drone in a particular direction is, Slow down the motor pointing in that direction and speed up the exactly opposite motor.

KINEMATICS

Vector data needs to be transformed from body fixed coordinate system of drone to ground coordinate system using some maths !
Any 2 configuration of a drone (rigid body) can be linked by a rotation about an axis. Further, Velocity and acceleration of any point on drone can be calculated (provided some data) using some equations.
One more multi-rotor drone model, Hexacopter is quite similar to quadcopter in motion principles (handling), and much better in stability and power. Diagonal motion can easily be performed due to 6 rotors.

Drone Dynamics

Drone Dynamics gives the governing equation that helps in controlling,automating and modelling the drone.
Dynamic quantities-

$$\underline{p} = M\underline{v}$$

$$\underline{L} = \underline{r} \times M\underline{v} + I\underline{\omega}$$

where

- p - linear momentum
- L - angular momentum
- M - mass of drone
- v - velocity of COM of drone
- I - moment of inertia of drone
- ω - angular velocity
- r - position of drone w.r.t reference point

The three laws that help in obtaining the governing equation are as follows-
●Mass balance - Drone is considered as a rigid body. The distance between any points on the drone remains the same. So there is no deformity and thus there is no change in mass.

●Linear momentum balance - Sum of all the forces acting on the drone is equal to mass of drone times the acceleration of COM i.e $\sum F=Ma$

●Angular momentum balance - $\sum m=\omega \times (I\omega) + I.\alpha$

where m- moment acting on drone and α - angular acceleration

Various motions of a drone are:

Thrust- Upward force produced by all motors of a drone.

Moving Forward (or Forward-Upward simultaneously)- By reducing power in front motors and increasing the same in back motors, this motion is achieved. Drone gets tilted and the rotor provides thrust to move forward, this motion is carried about the pitching axis.

Moving Right(or Left)-Can be done in the same manner as pitching motion but here drone tilts about ***rolling*** axis

Yaw- Yaw is controlled by turning up speed of an anticlockwise rotating motor and taking away power from clockwise rotating. Now, since clockwise torque is less, the quadcopter will rotate in CW.

PID Controller
❖A PID controller is used to regulate the position, orientation of our drone.

❖It compares the actual configuration with the desired configuration, computes the error and reduces it with the PID method.

What PID does with the error?
P : Proportion (Multiplies error with a constant)

I: Integral (Total of input so far)

D: Derivative (Current rate of change of error)

PID controller consists of three terms, namely proportional, integral, and derivative control. The combined operation of these three controllers gives a control strategy for process control. PID controller **manipulates the process variables like pressure, speed, temperature, flow, etc**.

MISSION PLANNER

Mission Planner is **a ground control station for Plane, Copter and Rover**.It is a fully-functioning GUI Ground Control Station(GCS) for multicopters, planes, helicopters & Rovers.

Functions of Mission planner:

- Planning the flight mission before flight
- Monitors and receiving flight information in Real time
- Configure the Autopilot according to our needs
- Virtual Simulating flights using SITL (using multicopters, planes, etc.)
- Configuring the RC Remote

SITL FEATURE-

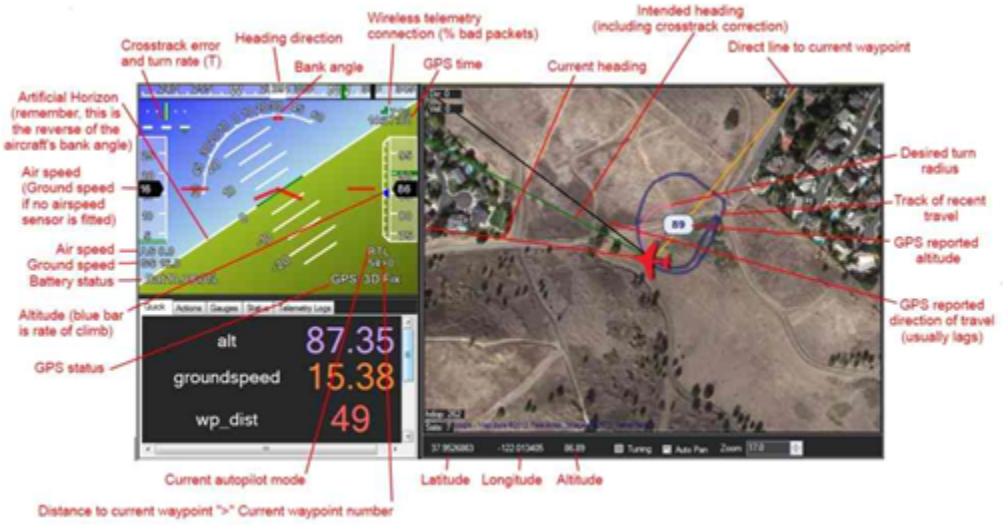
The SITL simulator allows to run a plane,copter or rover without any hardware.It works on ardupilot .It can be used to plan and execute missions.

FLIGHT PLANNING-

- This section covers what you see and what you can do when in the flight Plan screen of the mission planner .It allows us to manually scheme and execute the plan of the vehicle called missions.

-Any left mouse click adds a waypoint to the mission.

-Numerous options are available with a right mouse click such as inserting ,deleting waypoints(It is one of the most basic commands in mission planner .It guides the aerial vehicle towards a certain location with certain altitude),adding RTL,takeoff,jump,land,clear mission,etc.



SCREEN OVERVIEW OF MISSION PLANNER

Flight Modes available in Mission Planner :

- Acro Mode -Maintain altitude level
- Stabilize Mode:Allows manual control of vehicle
- Loiter Mode:Current altitude,orientation and heading direction is maintained.

Matlab and Simulink

MATLAB is a high-performance matrix-based language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. Typical uses include:

- Math and computation
- Algorithm development

- Modelling, simulation, and prototyping
- Data analysis, exploration, and visualization
- Scientific and engineering graphics
- Application development, including Graphical User Interface building

SIMULINK is a MATLAB-based graphical programming environment for modelling and simulating dynamical systems.Its primary interface is a graphical block diagramming tool and a customizable set of block libraries.

We were introduced to this programming platform through a series of onramp courses in both MATLAB and SIMULINK

Matlab Onramp Course

It serves as a great introduction to beginners who want to understand the features and workflows of the MATLAB environment
It introduces us to the basics of the MATLAB language which includes

- Commands
- Vectors and Matrices
- Importing Data
- Indexing and Modifying Arrays
- Array Calculations
- Calling functions
- Plotting Data
- MATLAB editor

Simulink Onramp Course

1. Course Overview: Running Simulations
2. Simulink Graphical Environment: Learn about Simulink blocks and signals
3. Inspecting Signals: Visualize signal values during simulation
4. Basic Algorithms: Use math and logic operators to write algorithms
5. Obtaining Help: Access documentation from Simulink

6. Project - Automotive Performance Modes: Practice working with math and logic operators
7. Simulink and MATLAB: Use MATLAB variables and functions in Simulink
8. Dynamic systems in Simulink: Review dynamic systems and learn how they relate to Simulink
9. Discrete systems: Model discrete-time systems
10. Continuous systems: Model continuous-time systems
11. Simulation Time: Choose the simulation duration
12. Project - Modelling a Thermostat: Practice your understanding of discrete dynamic systems
13. Project - Peregrine Falcon Dive: Practice your understanding of continuous dynamic systems

Some links where you can do the short, online onramp courses:

1. https://matlabacademy.mathworks.com/
2. https://www.mathworks.com/learn/tutorials/matlab-onramp.html

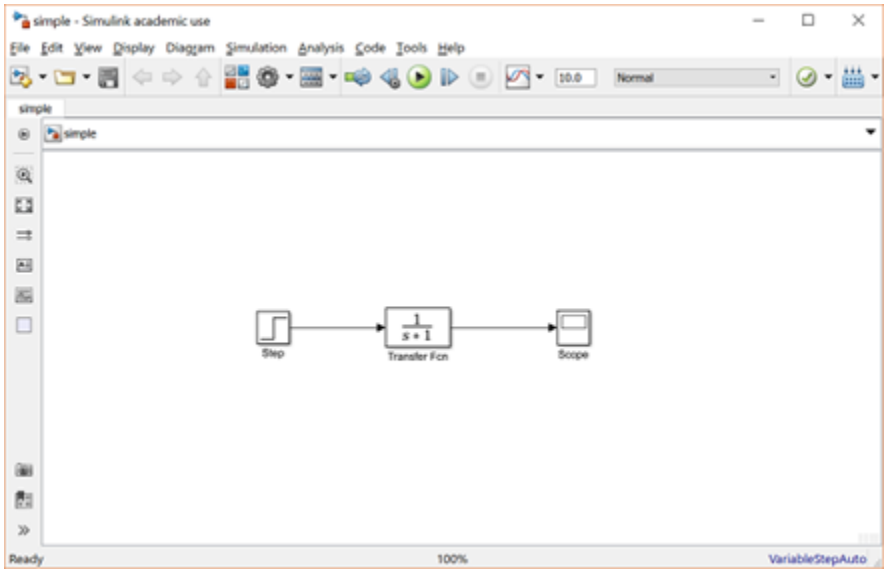
Drone Simulation

Drone Simulation is the behavioural modelling of a drone and evaluating its performance in a virtual environment. Simulation is an important part in the development of drones and helps us to:

- Understand drone dynamics and perform studies before building prototypes
- Tune parameters and models before uploading them to Drone
- Run various scenarios and test cases and algorithms without risking the Drone

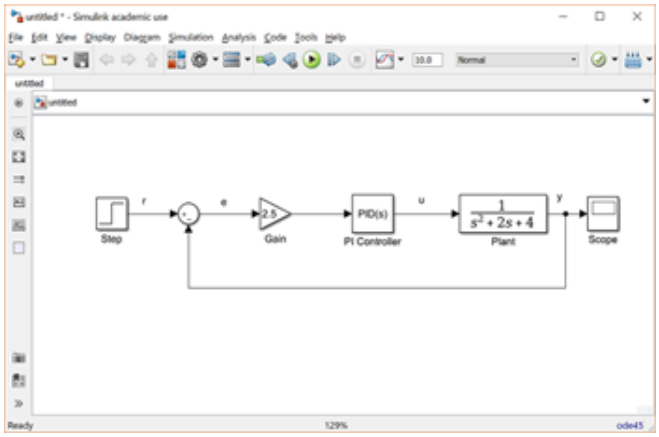
MATLAB and SIMULINK help us in making the components needed for Drone Simulation like –

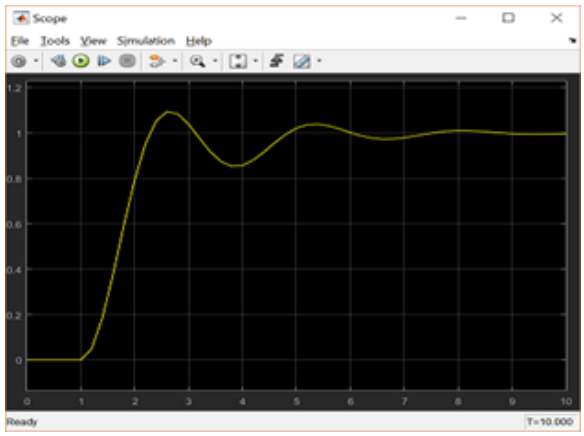
- Drone Dynamic model consisting of the drone's equations of motion
- Drone flight control model that models the control logic of the drone
- Sensor models simulating the sensors on the drone such as GPS and INS sensor
- Autonomous algorithms that perceive the environment and identify obstacles
- Simulation environments, such as Cuboid World and Unreal Engine, which are virtual environments created to test the algorithms and visualize the flight behaviour.



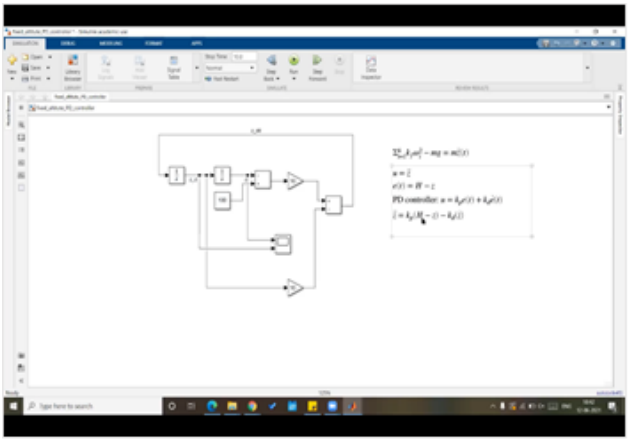
The simple model consists of three blocks: Step, Transfer Function, and Scope. The Step is a Source block from which a step input signal originates. This signal is transferred through the line in the direction indicated by the arrow to the Transfer Function Continuous block. The Transfer Function block modifies its input signal and outputs a new signal on a line to the Scope. The Scope is a Sink block used to display a signal.

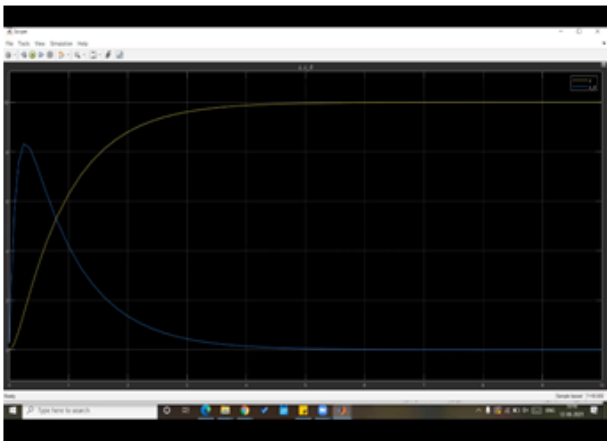
Another example of Simulink model-





Following are some visuals from live class-





UBUNTU Installation

For using ROS and Gazebo, we have to install Ubuntu 18.04 on our systems. The packages are small for download and the installation process is easy and can be done by the given links.

For the installation process, follow this link:
https://itsfoss.com/install-ubuntu-dual-boot-mode-windows/

Discord Server link for help:
https://discord.gg/ubuntu

GAZEBO

- ## What is gazebo ?

Gazebo is an open-source 3D robotics simulator.Gazebo integrated the ODE physics engine, OpenGL rendering, and support code for sensor simulation and actuator control.
A well-designed simulator makes it possible to rapidly test algorithms, design robots, perform regression testing, and train AI systems using realistic scenarios. Gazebo offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments.

- ## Uses-

- It makes it possible to rapidly test algorithms, design robots, perform regression testing, and train AI systems using realistic scenarios.
-Gazebo can use multiple high-performance physics engines, such as ODE, Bullet.
-It can model sensors that "see" the simulated environment, such as Laser rangefinders cameras (including wide-angle), Kinect style sensors.

- # Installing Gazebo:

Before starting, we have to make sure that we have gazebo installed. Gazebo originally comes with the ROS Distro.

- To confirm that you already have it installed, type the command

  ```
  $ gazebo --version
  ```

  In the command prompt, and check if it shows a version.
- Try the command

  ```
  $ gazebo
  ```

  This should open a new world.
- Let's get familiar with Gazebo by launching a 'husky' simulator. Type

  ```
  $ sudo apt-get install ros-melodic-husky-simulator
  ```

  If you are running on the Ubuntu 18.04 system.
- Setup an environment variable, any name will suffice(eg: HUSKY_GAZEBO_DESCRIPTION) by typing

```
$ export HUSKY_GAZEBO_DESCRIPTION=$(rospack find husky_gazebo)/urdf/description.gazebo.xacro
```

  Followed by

  ```
  $ roslaunch husky_gazebo husky_playpen.launch
  ```

  Which will launch a world, with a husky robot at the centre.
- For controls of husky, we will need to use teleop_twist_keyboard. For this type

  ```
  $ git clone https://github.com/ros-teleop/teleop_twist_keyboard.git
  $ cd ..
  $ catkin_make
  $ source devel/setup.bash
  ```

  In src directory

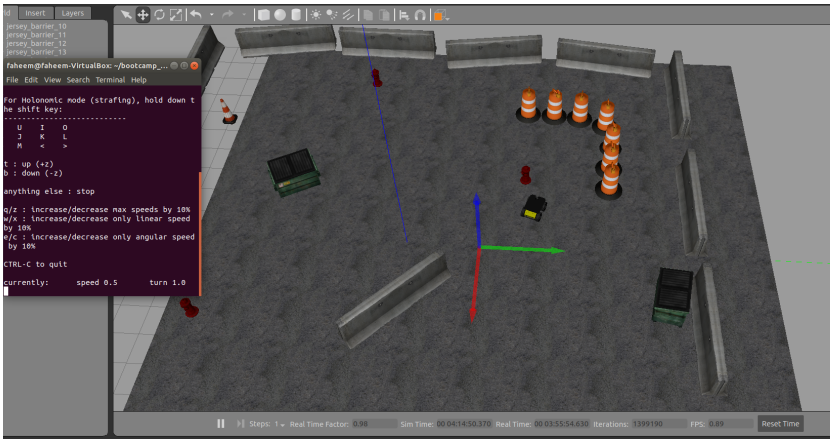
- Type

```
$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

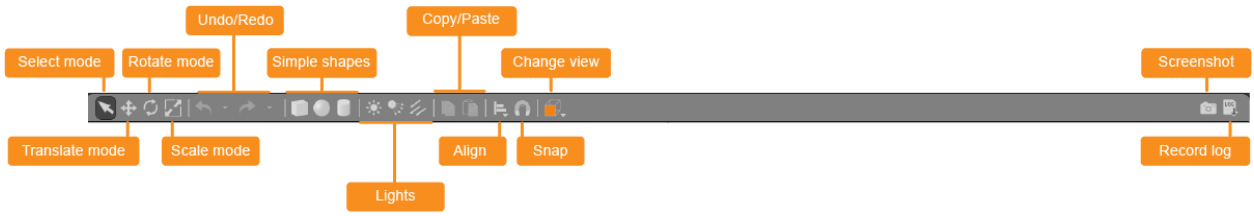
And then you can start with controlling the husky robot on Gazebo.

• USING GAZEBO:

- Once you have finished running the above commands, you will see the below screen:



- Navigate gazebo, use the options in the intended ways:



Function of different buttons in gazebo:

1. Select mode: Navigate in the scene
2. Translate mode: select models you want to move
3. Rotate mode: select models you want to rotate
4. Scale mode:select models you want to scale
5. undo/redo: Undo/ redo actions in the scene
6. Simple shapes: Insert simple shapes into the scene

7. copy/paste: Copy/paste models in the scene
8. Align: Align models to one another
9. Change view: View the scene from various angles

- GAZEBO in working:

ROS : Robot Operating System

- Introduction

The Robot Operating System (ROS) is a flexible framework for writing robot software. It is 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.

It is not a Language.

- Installation

  Reference Link: melodic/Installation/Ubuntu

- Why ROS?

❖ ROS is general

The same base code and knowledge can be applied to many different kinds of robots : robotic arms, drones, mobile bases, … Once you've learned about how communication is done between all the nodes of the program, you can set up new parts of an application very easily. In the future, if you need to switch to a totally different robotics project, you won't be lost.You'll be able to reuse what you know and some parts that you've built, so you'll never really start from scratch again.

❖ ROS has great simulation tools

ROS has many great tools, such as Rviz and Gazebo. With Gazebo one can even add some physical constraints to the environment, so when you run the simulation and the real robot, the outcome is pretty much the same. Imagine mapping a room in 3D with a drone directly on your computer, that could save you a huge amount of time.The simulation tools also allow you to see and use other robots that you don't possess, for educational purposes or to test in a specific environment.

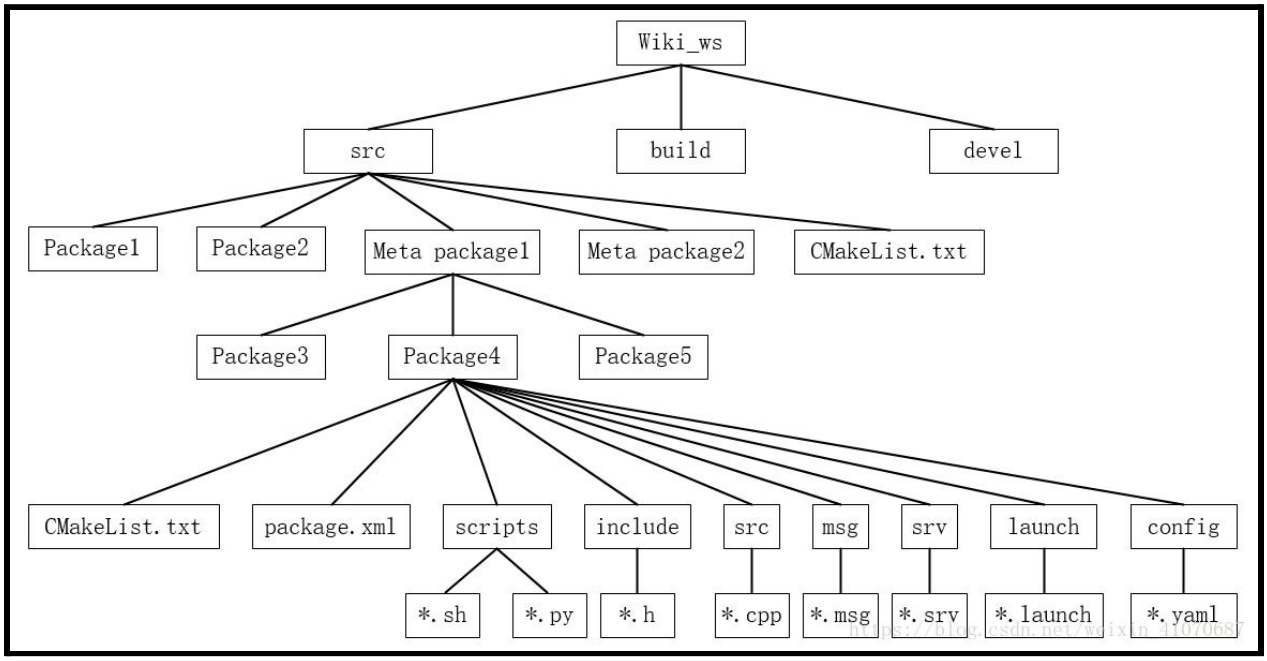
❖ Control multiple robots with ROS

ROS can work with multiple ROS masters. It means that you can have many independent robots, each with its own ROS system, and all robots can communicate between each other.

❖ ROS is an open source project

One of the greatest strengths of ROS is that it's open source.

❖ Structure of ROS : File System and Network



- **Nodes** : Nodes are processes that perform computation. ROS is designed to be modular at a fine-grained scale; a robot control system usually comprises many nodes. A ROS node is written with the use of a ROS client library, such as roscpp or rospy.
- **Topics** : The topic is a name that is used to identify the content of the message. A node sends out a message by publishing it to a given topic.
- **Services** : The ROS service is a kind of request/reply interaction between processes. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply.
- **Bags** : Bags are a format for saving and playing back ROS message data. Bags are an important mechanism for storing data, such as sensor data, that can be difficult to collect but is necessary for developing and testing algorithms.
- **Messages** : Nodes communicate with each other by passing messages. A message is simply a data structure, comprising typed fields.
- **Publishers/Subscribers** : A ROS Node can be a Publisher (Sender) or a Subscriber (Receiver) or both.

- **Packages** : Packages are the main unit for organizing software in ROS. A package may contain ROS runtime processes (nodes), a ROS-dependent library, datasets, configuration files, or anything else that is usefully organized together.
- **src** : This folder stores the C++/python source codes.
- **include** : This folder consists of headers and libraries that we need to use inside the package.
- **CMakeLists.txt** : This file contains the directives to compile the package.
- **package.xml** : This is the package manifest file of this package.

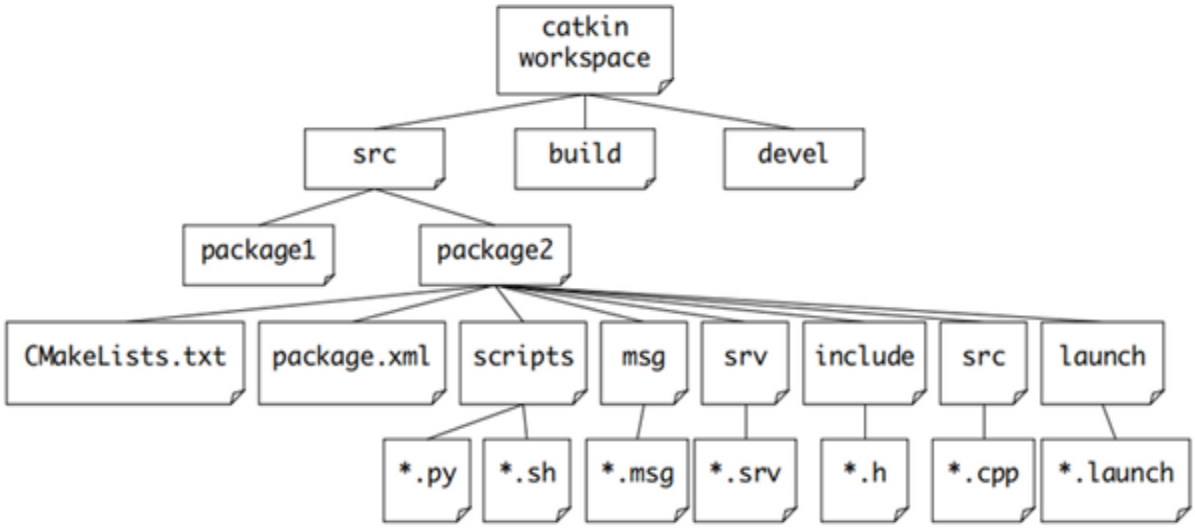
Catkin

It is the official build system of ROS and the successor to the rosbuild. It combines CMake and Python scripts to provide better functionality than CMake. It provides us a workspace to work with. All the ROS packages are clones in {catkin_ws}/src.

We need to install Catkin through the terminal command:

```
$ sudo apt-get install ros-melodic-catkin
```

The structure of Catkin workspace is as follows:



Catkin provides us with various command line tools for working with the catkin meta build-system and catkin workspaces. The various command line tools available in catkin are: -

- build – It is used to build packages in a catkin workspace
- configure – configures a catkin workspace's layout and settings

- clean- cleans products generated in a workspace
- create- create structures such as catkin packages
- env- run commands in a modified environment
- init- initialize a workspace
- list- find and list information about catkin packages in a workspace
- locate- get important workspace directory paths
- profile- manage different named configuration profiles

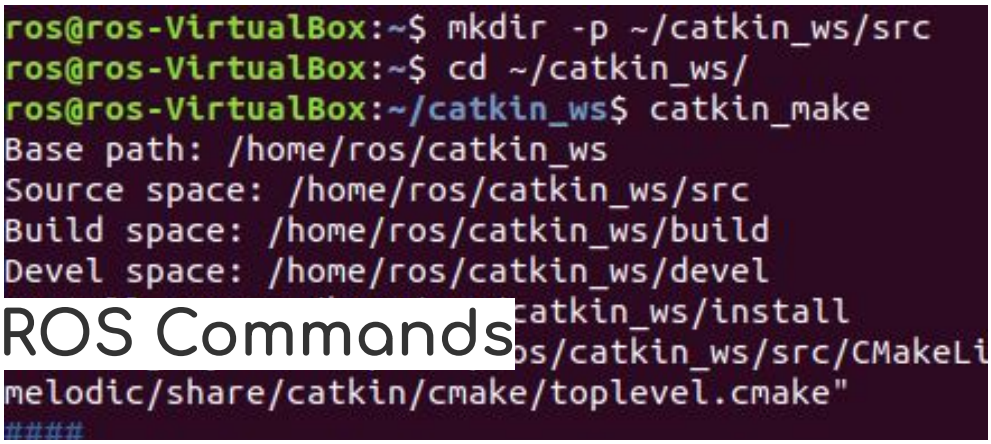
Install the Catkin tools using the following terminal commands: -

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu `lsb_release -sc` main" > /etc/apt/sources.list.d/ros-latest.list'
$ wget http://packages.ros.org/ros.key -O - | sudo apt-key add –
$ sudo apt-get update
$ sudo apt-get install python-catkin-tools
```

Creating a catkin workspace

- Create a folder, eg. catkin_ws. Move into the folder using the command $ cd bootcamp_ws and make a source folder with $ mkdir src.
- Now use $ catkin init inside it. Then run $ catkin build command to build the entire workspace.
- After this source i.e inform the system about package changes by $ source ~/catkin_ws/devel/setup.bash or you can add this line in the .bashrc file.

- To build a specific package use $ catkin build [PACKAGE_NAME].
- To clean the workspace use $ catkin clean.
- To check the status of the workspace use $ catkin config.



- ROS Commands

Commands	Action
roscore	This starts ROS.
rosrun	This runs an executable program and creates nodes
rosnode	This shows information about nodes and lists the active nodes
rostopic	This shows information about ROS topics
rosmsg	This shows information about the message types

rosservice	This displays the runtime information about various services and allows the display of messages being sent to a topic
rosparam	This is used to get and set parameters (data) used by nodes
roscd	used for switching between various ROS package directories
rosls	allows you to ls directly in a ROS package
rospack find	gives exact path to a ROS package
rosed	allows you to edit any file in any ROS package

ROSBAG

This is a set of tools for recording from and playing back to ROS topics. It is intended to be high performance and avoids deserialization and re-serialization of the messages.

It can record a bag, republish the messages from one or more bags, summarize the contents of a bag, check a bag's message definitions, filter a bag's messages

based on a Python expression, compress and decompress a bag and rebuild a bag's index.

Commands:

- record

    rosbag record subscribes to topics and writes a bag file with the contents of all messages published on those topics.

```
$ rosbag record <topic-names>
```

- play

    rosbag play reads the contents of one or more bag file, and plays them back in a time-synchronized fashion. Time synchronization occurs based on the global timestamps at which messages were received. Playing will begin immediately, and then future messages will be published according to the relative offset times.

```
$ rosbag play <bag-files>
```

    When you play a bag, no nodes are created,only the messages are published

ROS package:

The ROS software is organised into packages which may contain a collection of node executable files each performing a different task and many other files which will reduce the effort.
A ROS package typically consists of(apart from devel,build,log)

- src
- Cmake list
- package.xml (compulsory)

Additionally it may contain

- config
- launch
- include
- msg
- srv
- A custom named folder depending on your needs.

Src: This folder will contain all the node executable files (i.e the code for the task the node is specifically built for)
Launch: This folder contains launch files which are written in .xml format this is used to launch many node files at once along with additional advantages like viewing the parameter files and include some files necessary for launch purposes(For our case study it may be gazebo world or may be the quadrotor model etc).
Config: This contains all the param files written in .yaml format this is used to set parameters to allow code testing so that you don't need to change variables in each individual node.
Msg: This is a folder which contains definitions of structs(like geometry_msgs/Twist which contain 6 fields of information).This is very important and crucial for inter node communication
Srv: This folder contains the service definitions,these are .srv files and can not be performed frequently(like you can't publish messages frequently like

topic)Examples include in the turtlesim if you want to spawn another turtle you will need to call a service.

Include: This folder contains all the header files of C++.

1. packages are created by the catkin_create_package package_name [dependencies]
2. After which you can add your code files and edit the CMake lists and generate executables by catkin build(in case of roscpp) and by using chmod u+x .py in case of rospy written nodes
3. Then you have to source the devel space of the workspace you are working in so that the ROS master identifies it.
4. Then you can run node files by the rosrun package_name executable
5. Then you can run the launch file with roslaunch package_name [.launch file]

LAUNCH FILES

Launch files are very common in ROS to both users and developers. They provide a convenient way to start up multiple nodes and a master, as well as other initialization requirements such as setting parameters.

WRITING A LAUNCH FILE

Launch files are of the format .launch and use a specific XML format. They can be placed anywhere within a package directory, but it is common to make a directory named “Launch” inside the workspace directory to organize all your launch files. The contents of a launch file must be contained between a pair of launch tags

<launch> … </launch>

pkg/type/name: The argument pkg points to the package associated with the node that is to be launched, while “type” refers to the name of the node executable file. It is also possible to overwrite the name of the node with the name argument, this will take priority over the name that is given to the node in the code.

Respawn/Required: However optional, it’s common to either have a respawn argument or a required argument, but not both. If respawn=true, then this particular node will be restarted if for some reason it closed. Required=true will do the opposite, that is, it will shut down all the nodes associated with a launch file if this particular node comes down. There are other optional argument available on the ROS wiki.

ns: Another common use for a launch file is to launch a node inside a namespace. This is useful when using multiple instances of the same node. You can specify a namespace by using the “ns” argument.

arg: Sometimes it is necessary to use a local variable in launch files. This can be done using

<arg name=”…” value=”…”>

```
<?xml version="1.0"?>
<launch>
    <arg name="quad_name" default="quadrotor" />

    <include file="$(find interiit21)/launch/interiit_world1.launch"></include>

    <group ns="$(arg quad_name)">
        <node pkg="box_detector" type="box_detector_node" name="box_detector_node" output="screen">
            <rosparam file="$(find box_detector)/cfg/params.yaml" />
            <remap from="global_coord"    to="box_detector/global_coord" />
            <remap from="centre"    to="box_detector/centre_coord" />
            <remap from="odom"    to="/mavros/local_position/odom" />
        </node>
    </group>
</launch>
```

- First line: <?xml version="1.0" encoding="UTF-8"?> specifies that file is in XML
- Second line: <launch> specifies that file is a launch file
- Nodes are added using:

```
<node pkg="[PACKAGE_NAME]" type="[EXECUTABLE_NAME]"
name="[NODE_NAME]" output="screen"/>
```

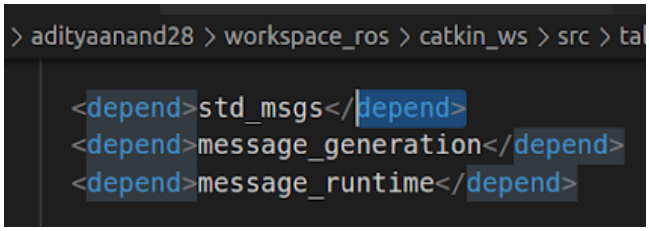
- The <group> tag makes it easier to apply settings to a group of nodes. It has an ns attribute that lets you push the group of nodes into a separate namespace. You can also use the <remap> tag to apply remap settings across the group.
- The <remap> tag allows you to pass in name remapping arguments to the ROS node that you are launching in a more structured manner than setting the args attribute of a <node> directly. The <remap> tag applies to all subsequent declarations in its scope (<launch>, <node> or <group>)
- The <include> tag enables you to import another roslaunch XML file into the current file. It will be imported within the current scope of your document, including <group> and <remap> tags
- The <rosparam> tag is reading in the rosparam yaml format from within the launch file. The value set by the <param> tag may only be a string, int, bool , or double , which may be set through the xml attribute value , or by reading in from a text file, bin file, or the output of a command line command.remap is used to change remap topic names

PACKAGE XML file:

It defines the properties of a package, like:

- Package name - <name> - name of the package.
- Version Number - <version> - required to be 3 dot-separated integer.
- Author - <maintainer> - name of the person managing the file.
- Dependencies (on other packages)

Of all these, dependencies is the most important one. It helps in exporting some needed data from other packages, which in turn helps in shortening the code a bit. Generally, 6 types of dependencies are used in common, these are:

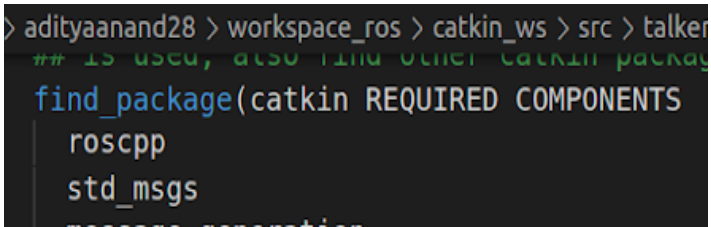


- <build_depend>
- <build_export_depend>
- <exec_depend>
- <test_depend>
- <buildtool_depend>
- <doc_depend>

Build dependency - Specify the packages which are needed to build this package.
Build export dependency - Specify which packages are needed to build libraries against this package.
Execution dependency - Specify which packages are needed to run code in this package.
Test dependency - Specify only additional dependencies for unit tests.
Build Tool dependency - Specify build system tool which the package needs to build itself.
Documentation tool dependency - Specify documentation tools which this package needs to generate documentation.

CMakeList txt file:

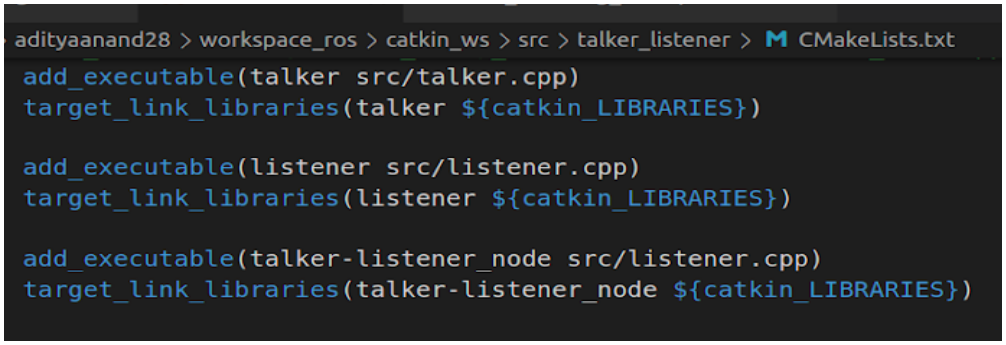
CMake is a build tool, i.e, build systems use it to do the building. Also, it is compiler independent, so it doesn't depend on the code language, other files are written in. it gave instructions like- libraries location, compiler to use, etc.



To add package dependencies and executables, we have to edit these:

- find_package()
- add_executables()
- target_link_libraries()

Some common dependencies are: rospy, sensor_msgs, nav_msgs, cv_bridge, etc.

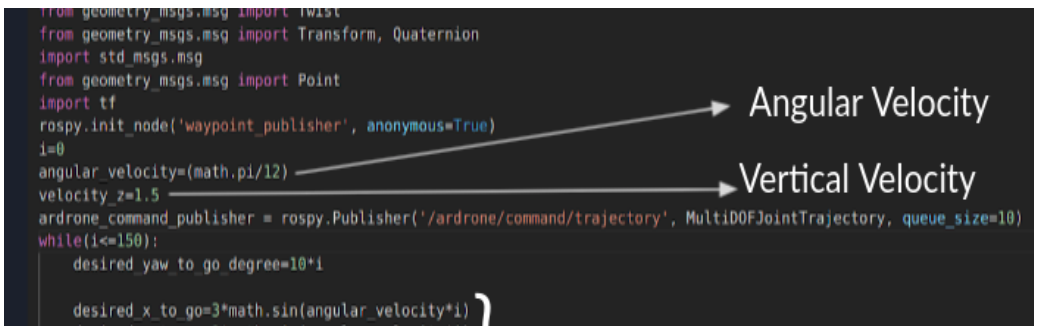


This way we can, add executable/ the programs to be run.

Find package - If a package is found by CMake through find_package, it results in the creation of several CMake environment variables that give information about the found package.
Executable Target - The programs we can run.
Library Target - Libraries that can be used by executable targets at build and/or runtime.

RotorS - UAV gazebo simulator:

This is a done setup gazebo model. We selected to go with a helical path for the model. We gave angular and vertical velocity to the quadcopter, and set up a target location, at which it stops after coming at that point.

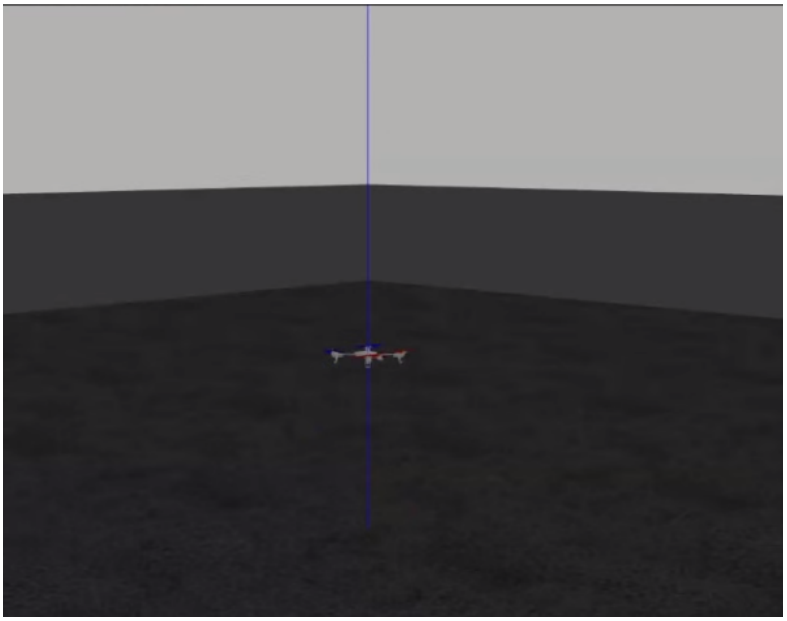


Like this, we set up the parameters for the helix path.

```
:~$ roslaunch rotors_gazebo mav_with_waypoint_publisher.launch mav_name:=ardrone world_name:=empty
```

We created the package and built it using the catkin build function in the catkin workspace. For launching the required model, we ran roslaunch (given in the figure above)in a terminal, and simultaneously, the following in another tab.

```
tejas@tejas-Surface-Laptop:~$ rosrun drone_publisher waypoint_publisher.py
```



This was the final output, after running the launch file, which further started hovering at the desired location after reaching there.

Blue line is the line of axis, about which it forms a helical pattern as you can see in the video we presented

Reference Links:

Video:

-drone RotorS video

https://drive.google.com/folderview?id=18UqLZu0rVxwtQLuGwaV5XJ5ZpTCQXjxn

workspace and turtlesim

https://drive.google.com/file/d/125Z6gsszC9-aQY2DZlyUxw0NIbRlsfgZ/view?usp=drivesdk

-husky play pen

(https://drive.google.com/file/d/1cVxCCYzzUcKK_Kqfb_wfYSN1lBGP0lM_/view)

-robocup working video

(https://drive.google.com/file/d/1kGK4s0k5mOkYctGQ0AuHSC2qLBrEQ_yF/view)

-Willow garage

(https://drive.google.com/file/d/1kGK4s0k5mOkYctGQ0AuHSC2qLBrEQ_yF/view)