

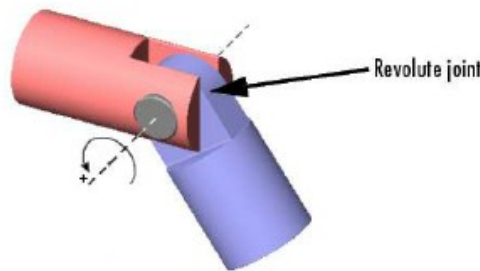
ROBOTIC ARM SIMULATION – DOCUMENTATION

1) FORWARD KINEMATICS

Introduction:

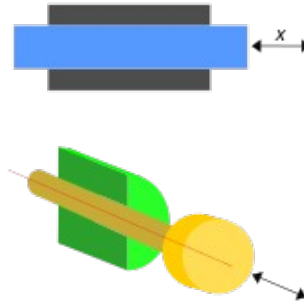
Robotic arms in more general terms can be referred to as kinematic chains. A kinematic chain is a combination of links that are connected together by joints. Joints are generally modelled to have a single degree of freedom. A **degree of freedom**, in the kinematics context, refers to the number of independent variables/co-ordinates necessary to completely describe the position of a system in space. In case a joint is seen to have more than one degree of freedom, we model that joint as a combination of multiple joints have one degree of freedom each. The different types of joints are as follows:

Revolute joint:



Revolute joints (also called pin joints or hinge joints) are single degree of freedom kinematic pairs, which can rotate (or hinge) about a single axis. Ex: in door hinges & folding mechanisms. They can be used as a passive or an active (i.e., driven by a motor) joint.

Prismatic joint



Like revolute joints, prismatic joints also have a single degree of freedom. However, the difference between the two lies in the type of motion they permit. Prismatic joints are used to define translational motion along an axis or sliding motion along an axis (hence sometimes referred to as a slider joint). Here rotational motion is completely restrained. Ex. Slider in a slider-crank mechanisms. They can be used as both passive and active joints (i.e., driven by a motor).

Joint value: In order to define a joint, we use a joint value. It can either be the angle by which rotation has taken place about the joint axis (in case of a revolute joint), or the amount of translation that has occurred along an axis (in case of a prismatic joint).

In the context of robotic arms, we need to figure out the values of the several joints that are present in the kinematic chain, so that the end effector ends up at a specific point in space. Thus, in forward kinematics, we determine where the end effector will be placed in 3D space given certain specific joint values for all the joints in the system under consideration. However, this doesn't mean we don't consider the positions of the other links. (For example, we need to ensure that the elbow of a robotic arm doesn't collide with any obstacle in the process of getting the end effector to a certain position in space). Each link has an associated coordinate frame associated with it. We also need to determine what the transform is from the base link of the robot to the end effector link of the robot. In most cases, industrial robots have joint encoders, and thus the robot's firmware will let the user know the joint values.

In summary for FK,

- **Input:** Joint angles

- **Output:** Co-ordinates of end effector / Transform from base to end effector

- Transforms for the links depends on the design of the link and therefore doesn't change as long as the design of the link doesn't change

- Transforms for the joints depend on the joint angles, that continuously changes during run time of the robotic arm, and thus it is a variable.

INVERSE KINEMATICS:

In inverse kinematics, we will already know where the end effector of the robotic arms needs to be. In other words, we will already know what the transform from the base to the end effector must be. Using this, we will need to calculate the joint values. It is therefore, as the name suggests, the opposite of forward kinematics.

- **Input:** Co-ordinates of end effector / Transform from base to end effector

- **Output:** Joint angles

Robot description: There are two notations that we use to describe a robot:

1) **URDF** – This is an abbreviation for universal robot description format, and is typically used in research. It is not as common in the industry. The reason is that since URDF is quite descriptive and lengthy, the storage space it consumes is a problem. Also, it supports solving for FK and IK using numerical techniques rather than analytical techniques. During the industrial revolutions when automation of processes in factories and industries started booming, we did not have devices with the storage capacity that today's devices have. Therefore, numerical techniques of solving, which required space as well as a lot of computational power was not preferred. As a result, URDF was not very popular. However, with increasing computational power and storage capacity in to, this will not be a problem in the future, and we can see an increasing dominance of URDF.

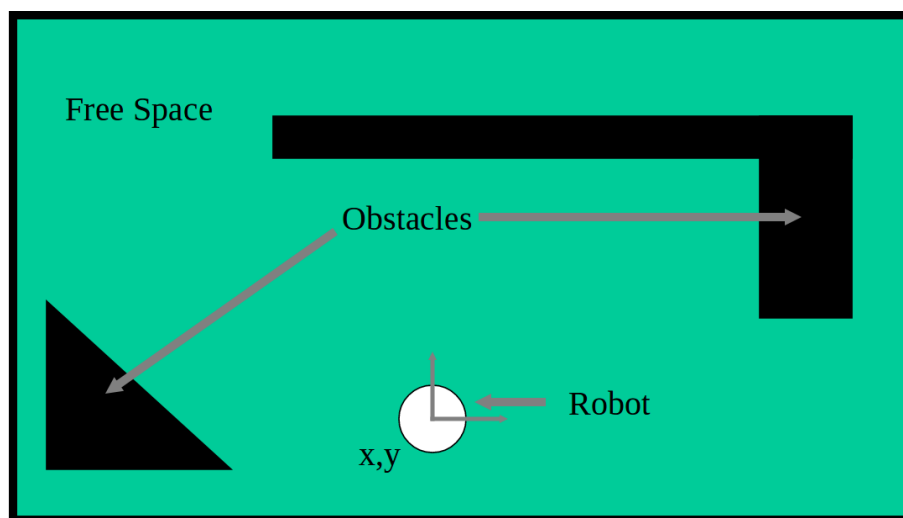
2) **DH** – This is an abbreviation for Denavit-Hartenberg notation. This is used by industrial robots and is more compact in its representation and also supports analytical solutions to FK and IK problems. It works on a set of assumptions defined and known by the robot manufacturer and the user. This is gradually being replaced by URDF as the computational power and storage capacity of devices are increasing.

MOTION PLANNING

The world in which a robot is placed can be seen to be composed of two distinct features:

(i) Obstacles (i.e., regions of space in the world that have been occupied. In other words, a robot cannot go there)

(ii) Free space (i.e., regions of space where a robot ‘might’ be able to go).



There are many different ways to measure how good a path is: (i) Time (ii) Distance (Iiii) Cost ..etc.

Motion planning is a process of breaking down a desired movement task into a set of smaller discrete motions, which satisfy a set of constraints and also optimise certain aspects of motion using parameters as discussed above (to define “how good” a path is). The basic motion planning problem is to determine a continuous path between a starting point and the final destination to which the robot needs to go. The robot and obstacle geometry is described in a 2D/3D workspace and the path is described in a “configuration space”.

A “configuration” describes the pose of the robot and the “configuration space” describes the set of all possible poses that a robot can be in. There are several algorithms which are used for motion planning including grid based search, interval based search, geometric algorithms, reward-based algorithms, A*, D* etc., each with a set of merits and demerits. The selection of an algorithm depends on the complexity of the problem, computational power available, and time available to solve problem.

Robot Operating System (ROS)

An Introduction to Robot Operating System (ROS):

Robotics is a field which is gaining incredible traction over the last few years and a wide range of robots are being developed for a diverse range of applications. As the scale and scope of applications of robotics continues to increase, writing robust software which can be reused becomes significant.

ROS is a flexible framework which simplifies the process of creating robust robotic behaviour across a diverse range of robotic platforms. A **framework** is a collection of programs that do something useful and which you can use to develop your own applications. ROS by itself adds a lot of value to robotics projects. In addition to this, it also allows collaboration with world class roboticists who are a part of the ROS community. ROS is an open source project. Therefore, the code within it is the result of collaboration of this international community of roboticists. There are many ROS distributions available for use of the community. Some are older releases with long term support (aka LTS), making them more stable, while others are newer with shorter support life times. For these reasons, the ROS firmware that is being used for this simulation is ROS Kinetic Kame.

ROS currently requires Ubuntu to run. A common question is that since ROS, as the name claims, is an operating system, what is the necessity to have Ubuntu to run ROS? Technically, ROS is not strictly an operating system, but a “meta”-operating system. This means that ROS provides services that a typical operating system will provide, but cannot be stand-alone. It is run on top of another operating system, in this case Ubuntu.

ROS TERMINOLOGY

Nodes:

A ROS node is any executable program that is required to perform some computation in our robotic system. Let us understand this by taking the example of a simple line follow robot. There are two main components present in the line follow robot, which are the sensors (let's assume we use one IR sensor), and motors. To simulate this robot on ROS, we need nodes or executable programs, which will perform different functions. One node will perform the function of getting sensor data from the IR sensor, the other node uses the sensor data to ensure that the path of the robot is along the line it needs to follow, and another will be responsible for actuating the motors of the wheel. You can observe that all the nodes mentioned above have a specific function in the operation of the robot, and they also need to communicate with one another for proper functioning of the robot. For this we need to ensure a proper system for ROS nodes communicate with one another. This communication of information between nodes is done with the help of ROS topics.

Topics:

A ROS topic is a stream of messages which is sent by a node. Any node which 'sends' or 'publishes' data is called a **publisher**. The data which is being published by a node can be 'received' or 'subscribed' by another node, called the **subscriber**. You can find the links for how to write a basic publisher and subscriber in my github page: https://kaushikbalasundar/ros_pubsub.git
We will be using the python scripts to explore topics in the next section once you have your workspace set up.

Some basic commands that you can execute in your terminal with regard to ROS topics are as follows:

```
>> rostopic list : lists the active topics
>> rostopic echo /topic : print information about the contents of a topic
>> rostopic type //print topic type
>> rostopic hz //print publishing rate
```

Packages:

Any software we write in ROS needs to be organised into a package. Each package can contain libraries, executables, scripts, etc. All the nodes, as described above, must be placed inside a package of its own. The ROS build system is called catkin. The catkin build system will be explained in detail in the next section. Follow these instructions to create a new package:

1. Create a new directory called `catkin_ws` with a sub directory called `src` within it using the following command:

```
>> mkdir -p /catkin_ws/src  
>> ls
```

On typing the `ls` command from your home directory in your terminal, you will be able to see that the `catkin_ws` folder has been created. All your ROS work will happen within this folder.

2. Change your working directory to `catkin_ws` using this command:

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

3. Next, we need to initialise the location so that a symlink can be created. Use this command:

```
>> catkin_init_ws
```

4. Change directory back into the catkin workspace and build your workspace using this command:

```
>> catkin_make
```

The process will take some time, so be patient.

5. After building it, we need to set the environment of the workspace so as to make it visible to the entire ROS system. Use this command:

```
>> echo source ~/catkin_ws/devel/setup.bash >> ~/.bash.rc  
>> source ~/.bashrc
```

This process is done to tell the terminal where different ROS packages are located and also to update environment variables. The terminal by default will not know what commands like `roslaunch`, `rostopic list`, etc., means. To ensure that it does, we perform the above step. With this your workspace is set up and ready to go.

6. Open your `catkin_ws` folder. You should notice that in addition to `src`, there are now two additional folders called `'build'` and `'devel'`. The `'src'` folder is where all your packages will be placed, be it the ones you create or those that you clone from github.

7. To create our package, inside `src`, change directory to source :

```
>> cd src
```

8. To create a new catkin package, use the following command:

```
>> catkin_create_pkg hello_ros roscpp rospy std_msgs
```

Here, the name of the package is called `'hello_ros'`, and it is followed by the dependencies of the package. Dependencies tell us what kind of files can be used within the package. In this case, we are giving compatibility to C++ (`roscpp`) and Python (`rospy`) files, along with messages (`std_msgs`) that will be contained within the ROS topics that are published or subscribed by the nodes we create in C++ or Python.

9. Open your `hello_ros` folder and you will see folders such as `src` & `launch`. To run python files, create a new folder called `'scripts'`, and to create launch files, they need to be in a new folder called `'launch'`. Create a new folder called `'launch'`. This naming convention is universal in ROS and has to be followed.

10. Set up a git hub account by following the instructions at the end of the document. Clone the repository into your `scripts` folder:

```
>> cd ~/catkin_ws/src/hello_ros/scripts
```

```
>> git clone https://github.com/kaushikbalasundar/ros_pubsub.git
```


Ensure that the listener.py and talker.py python files are in the folder called scripts. We will see how to run these files in the coming section when rosrn is introduced.

10. Navigate back to catkin_ws and run catkin_make:

```
>> catkin_make
```

It is important to run catkin_make after any changes are made within your workspace.

Manifests:

A manifest is a description of a package. This terminology is not restricted to ROS, but other platforms also have the same nomenclature for such packages. It defines dependencies between packages, and to capture meta-data (meta data is basically any information about some data) about a package such as version, licence, etc.

Roscore, rosrn, roslaunch:

The roscore command runs a set of nodes which need to be run mandatorily to run other nodes that we have written. Nodes such as ROS master, ROS parameter server, and ROS out logging node are necessary to run any other executable node that you have written. ROS Master has the complete data about the network of nodes and is like a communication coordinator, which is used to coordinate the transfer of information from publishers to subscribers. In order for us to run the files called talker.py and listener.py, we will need to use a command in ROS called rosrn. It is a command which is used to run one node and has the following syntax:

```
>> rosrn <name of package> <name of file>.
```

Note: Open a new terminal and type rostopic list to see the list of topics being published. If there is no roscore running, you will get the following error: ERROR: Unable to communicate with master! This is because there is no ROS master to coordinate the communication between nodes, and so there are no topics being published. Open another terminal and type:

```
>> roscore
```

Again in the previous terminal, type:

```
>> rostopic list
```

You will now see that two topics /rosout and /rosout_agg are now being published as roscore is running. Also, we now have a ROS master running and other nodes can be executed. Let us now run the nodes talker.py and listener.py. Follow these instructions:

1. In a new terminal, type:

```
>> roscore
```

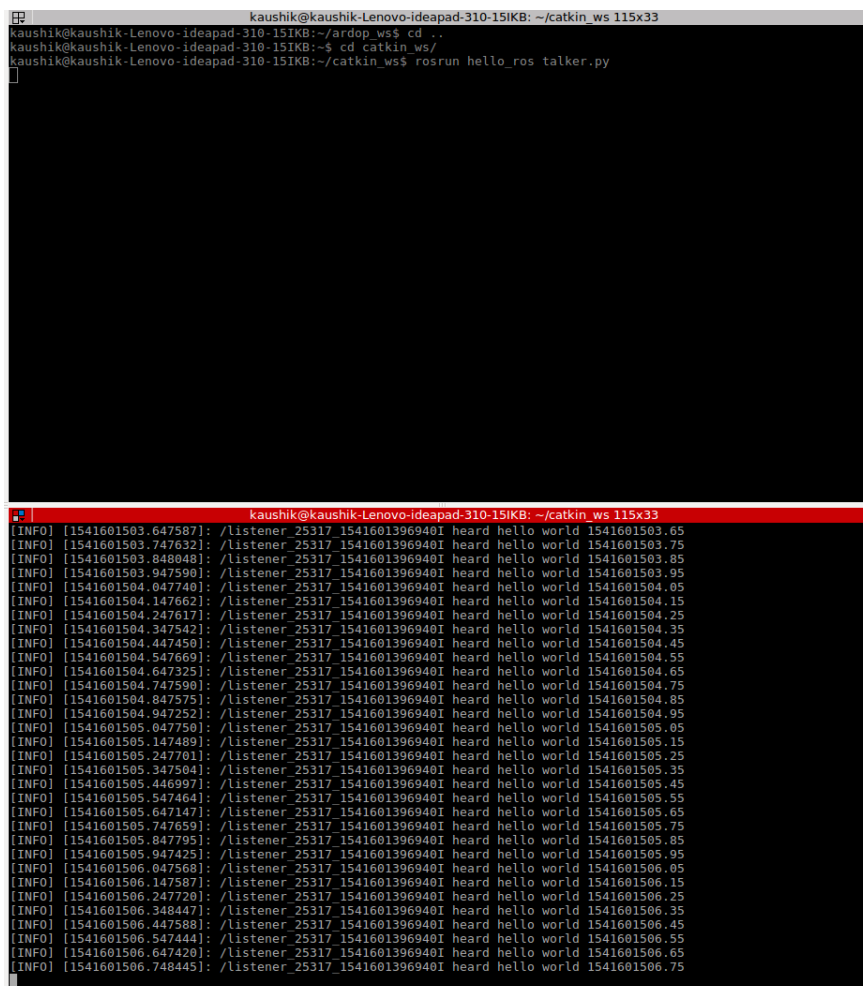
2. Change directory to catkin workspace and type the following commands:

```
>> cd ~/catkin_ws/src/hello_ros/scripts
```

```
>> rosrun hello_ros talker.py
```

This will now run the node called talker.py, which will publish messages under the topic called /chatter as described in the script.

4. Open another terminal and type `roslaunch hello_ros listener.py`. This will run the subscriber node which will subscribe to the topic called /chatter. The output will look like this:



```
kaushik@kaushik-Lenovo-ideapad-310-15IKB: ~/catkin_ws 115x33
kaushik@kaushik-Lenovo-ideapad-310-15IKB:~/ardop_ws$ cd ..
kaushik@kaushik-Lenovo-ideapad-310-15IKB:~$ cd catkin_ws/
kaushik@kaushik-Lenovo-ideapad-310-15IKB:~/catkin_ws$ rosrun hello_ros talker.py

[INFO] [1541601503.747632]: /listener_25317_1541601396940I heard hello world 1541601503.75
[INFO] [1541601503.848048]: /listener_25317_1541601396940I heard hello world 1541601503.85
[INFO] [1541601503.947590]: /listener_25317_1541601396940I heard hello world 1541601503.95
[INFO] [1541601504.047740]: /listener_25317_1541601396940I heard hello world 1541601504.05
[INFO] [1541601504.147662]: /listener_25317_1541601396940I heard hello world 1541601504.15
[INFO] [1541601504.247617]: /listener_25317_1541601396940I heard hello world 1541601504.25
[INFO] [1541601504.347542]: /listener_25317_1541601396940I heard hello world 1541601504.35
[INFO] [1541601504.447450]: /listener_25317_1541601396940I heard hello world 1541601504.45
[INFO] [1541601504.547669]: /listener_25317_1541601396940I heard hello world 1541601504.55
[INFO] [1541601504.647325]: /listener_25317_1541601396940I heard hello world 1541601504.65
[INFO] [1541601504.747590]: /listener_25317_1541601396940I heard hello world 1541601504.75
[INFO] [1541601504.847575]: /listener_25317_1541601396940I heard hello world 1541601504.85
[INFO] [1541601504.947252]: /listener_25317_1541601396940I heard hello world 1541601504.95
[INFO] [1541601505.047750]: /listener_25317_1541601396940I heard hello world 1541601505.05
[INFO] [1541601505.147489]: /listener_25317_1541601396940I heard hello world 1541601505.15
[INFO] [1541601505.247701]: /listener_25317_1541601396940I heard hello world 1541601505.25
[INFO] [1541601505.347504]: /listener_25317_1541601396940I heard hello world 1541601505.35
[INFO] [1541601505.446997]: /listener_25317_1541601396940I heard hello world 1541601505.45
[INFO] [1541601505.547464]: /listener_25317_1541601396940I heard hello world 1541601505.55
[INFO] [1541601505.647147]: /listener_25317_1541601396940I heard hello world 1541601505.65
[INFO] [1541601505.747659]: /listener_25317_1541601396940I heard hello world 1541601505.75
[INFO] [1541601505.847795]: /listener_25317_1541601396940I heard hello world 1541601505.85
[INFO] [1541601505.947425]: /listener_25317_1541601396940I heard hello world 1541601505.95
[INFO] [1541601506.047568]: /listener_25317_1541601396940I heard hello world 1541601506.05
[INFO] [1541601506.147587]: /listener_25317_1541601396940I heard hello world 1541601506.15
[INFO] [1541601506.247720]: /listener_25317_1541601396940I heard hello world 1541601506.25
[INFO] [1541601506.348447]: /listener_25317_1541601396940I heard hello world 1541601506.35
[INFO] [1541601506.447588]: /listener_25317_1541601396940I heard hello world 1541601506.45
[INFO] [1541601506.547444]: /listener_25317_1541601396940I heard hello world 1541601506.55
[INFO] [1541601506.647420]: /listener_25317_1541601396940I heard hello world 1541601506.65
[INFO] [1541601506.748445]: /listener_25317_1541601396940I heard hello world 1541601506.75
```

5. Open another terminal, and type:

```
>> rostopic list
```

See the new topics which are now added to the communication system.

6. For visual representation of the topics being published and subscribed by various nodes, type:

```
>> rqt_graph
```



This will give you a visual representation of all the topics involved in the communication between different nodes. This is particularly helpful for debugging.

Note:As you can see, we will require the use of many terminals at once to execute our nodes. For this, a software called ‘Terminator’ can be installed which will allow you to run multiple terminal screens on the same window.

Rosrun allows us to run one node at a time. However, in most cases, we will need to run multiple nodes at the same time for a particular application. For example, to run the gazebo simulation of a robotic arm, we will need to run multiple nodes. Running these nodes individually is a time consuming process and is not practical. Therefore, we make a launch file in which multiple ros nodes can be called using a single command. All the launch files are stored inside a folder called launch inside your package. Running a launch file will automatically run roscore and the start the ROS master. As a result, we don’t need to open a separate terminal and run roscore explicitly. In a single command, all our required nodes will be run. To run the gazebo simulation to control all the joints of our simulation, use the following command after cloning the file into your src folder of the workspace.

```
>> cd ~/catkin_ws/
```

```
>> roslaunch ardop_description position_controllers.launch
```

The generalised format for running roslaunch is as follows:

```
>> roslaunch <package_name> <name of launch file>
```

All launch files are written using .xml format.

Using <remap> functions:

Let us take the example of the simple publisher and subscriber python scripts we referred to earlier when discussing rosrn. The topic published by the talker was called /chatter and the topic subscribed by the listener was called /chatter. Hence, there was no problem in communication and it occurred properly as shown by the rqt_graph. However, sometimes the listener might subscribe to a different topic, say /singing while the talker is publishing a topic called /chatter. We need to tell the ROS master that both these topics are one and the same, so that any time a node comes to the ROS master asking for a topic called /singing, it will direct it or 'remap' it to /chatter. This is a powerful tool because we don't have to change the names of topic in each subscriber node individually, but we can do it in one go inside the launch file.

Catkin build system:

Before we understand what catkin is and what it does, let us first address what a build system does. "Build" is an activity to translate human readable source code into an efficient executable program. A set of programs that facilitate this process is called a build system. "catkin" is the official build system of ROS and has replaced the original ROS build system, which was called "roscbuild". catkin combines both CMake macros and Python scripts. To understand what this means, let us take a look at what Cmake does. Cmake is a build system, which means that it helps in compiling the code correctly, by performing functions such as linking libraries that we used in our files. In other words, it converts human understandable, high level code into more efficient executable 'make' files (hence called Cmake). Since catkin uses Cmake, workflow is very similar to CMake's but adds support for automatic 'find package' infrastructure and building multiple, dependent projects at the same time.

As mentioned earlier, catkin_ws is organized into three distinct folders: 1) Src (Any code we write will be placed here inside their respective packages) 2) Devel (Some environment setup files are located here, which aren't to be tampered with) 3) Build (All executable version of the code we wrote we will be stored here. It contains cache information and other intermediate files).

Git:

Git is a system which allows software development to take place such that multiple users can work on the same software. It allows coordinating and controlling the changes made to the source code and only the best code, as agreed by all the users working on a project. Every Git directory on every computer is a full-fledged repository that contains the history and full version-tracking abilities. Let us set up github on our systems. Follow these instructions to install git and configure your account:

1. Make an account on <https://github.com>

2. Install git in your system:

```
>> sudo apt-get install git
```

3. Next, configure it for your username using the following command:

```
>> git config --global user.name " <type your username here within the quotations> "
```

```
>> git config --global user.email " <enter your registered email address here> "
```

To clone any repository from github into your computer (cloning is a process of copying the files from github so that you can run the code and make any changes to it on your computer):

1. Open a new terminal and navigate to the src folder inside your workspace :

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

2. Clone the github repository using this command:

```
>> git clone https://github.com/kaushikbalasundar/robotic\_arm\_simulation
```

3. Change directory back into your workspace and run catkin_make to update the changes made inside src:

```
>> cd ..  
>> catkin_make  
>> source devel/setup.sh
```

Uploading your codes to git using command prompt:

1. To upload any files to Github using the command prompt, create a new repository in your github account and make sure to select the check box for creating a README.md file. Ensure that your repository has the same name as the folder in which all the codes you want to upload is located.

2. Open a new terminal and change directory to the folder that contains all the code you want to upload onto github:

Example:

```
>> cd ~/ardop_ws/src/
```

3. Initialise git in this directory:

```
>> git init
```

4. To add all the files in the current folder you're in, use the following command

```
>> git add .
```

Alternatively, if you want to add only a specific file:

```
>> git add <name of file>
```

5. To submit your changes onto github, (in other words, also called committing your changes), use:

```
>> git commit -m "First commit" //create first commit
```

Note: Add an appropriate message indicating what changes you have made to the existing folder when you commit new changes.

6. Next, configure git to use the repository you created in step 1 to push all your code into. You only have to execute the following command once:

```
>> git remote add origin https://github.com/<your_username>/<repository_name>
```

7. Finally, to push all your code into the master branch of your repository, use:

```
>> git push origin master //push the commit into master
```

Note: This command will ask you to enter your username and password at this stage. Do so as intimated by the command prompt. Once your credentials are verified, the code will be pushed onto your github repository.

Here we commit directly into master as we have set up only one branch. The master branch is at the highest level of hierarchy. If multiple users are working on different versions of the code, multiple sub branches are created and their code is committed to their respective branches. When users agree a particular branch has the best version of the code, it is migrated up to the master branch, by committing directly to master. In my case, since I am the only contributor to the code and I have only one version, I have only the master branch set up.