RBE-500
GROUP ASSIGNMENT-1
Authors: Philip Lund, Muqeet Jibran Mohammed Abdul

The report is organised as follows:

The first 3 sections deal with solving problems 2 and 3 of the assignment, and sections 4 and 5 explain implementation of the 1st Problem. THe conclusion summarizes the work done and the next steps. Important commands: **(i)** To **launch gazebo (is in Section V),** to **(ii) get the Inverse kinematics service and call it (in Section III)** and to **(iii)** run forward kinematics after launching the file, just navigate to fwdkin.py and run it (see file tree in Section V) or use **rosrun rbe_proj fwdkin.py.**

(I) DH-BASED CALCULATIONS

In this project, a SCARA robot was modelled. First, the Denavit-Hartenberg (DH) Convention was used to create a series of axes for the robot's links and axes(Figure 1). From there, a DH table was constructed, allowing for the forward kinematics to be calculated by multiplying the various homogeneous transforms obtained from them (Figure 2). In order to calculate the inverse kinematics, a geometric strategy was used (Figure 3 and Figure 5).

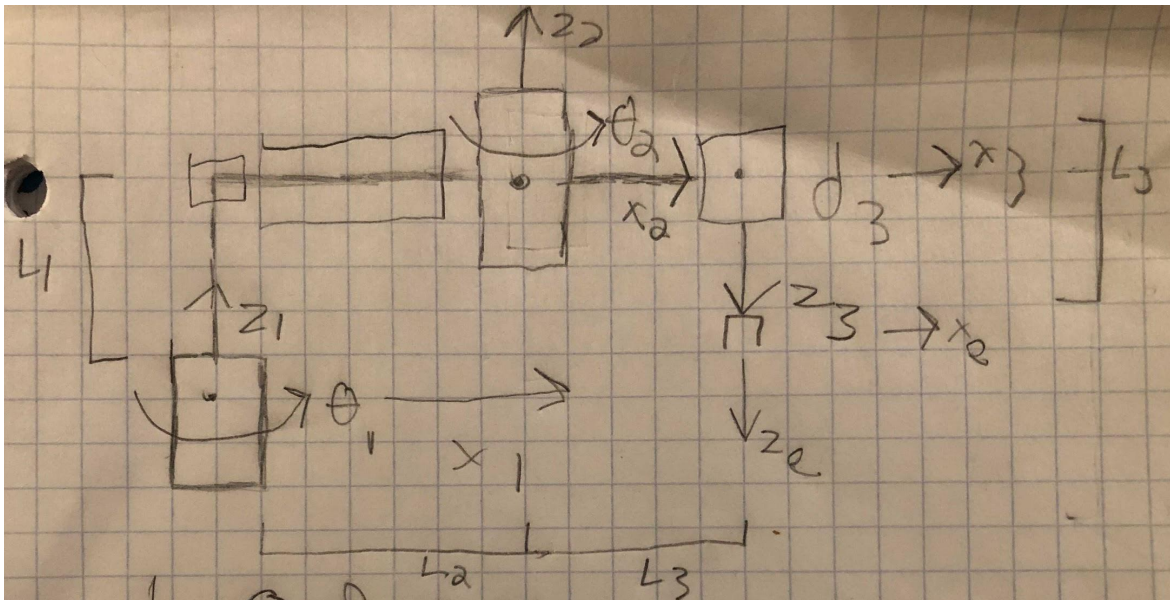

Figure 1: The axes assigned to each link, as well as lengths and placeholder blocks for the model

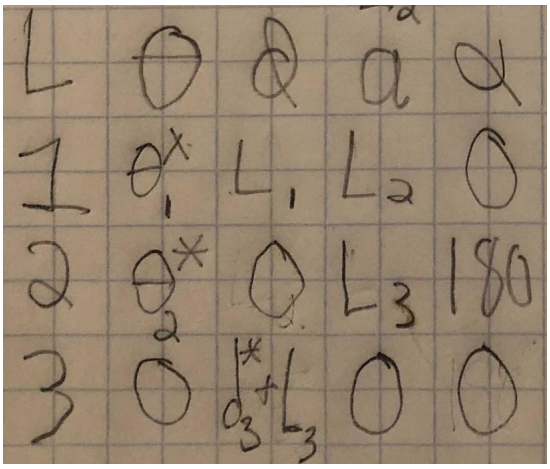

Figure 2: The DH Table derived for the SCARA manipulator

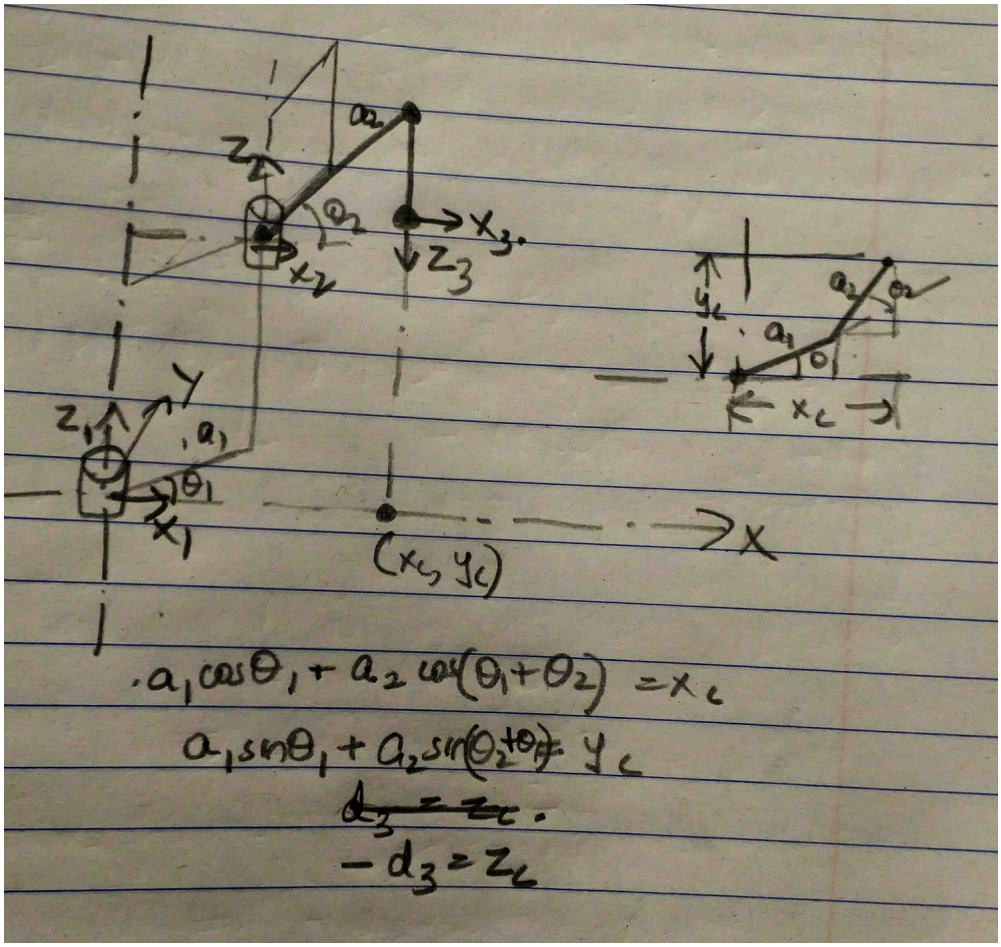

Figure 3: The contribution of joints to end position. For the inverse kinematics of first 2 joints, a method similar to the typical 2 link arm was used(See Fig. 5). The third joint, which is prismatic, contributes to z-position.

(II) FORWARD KINEMATICS

In order to convert these calculations into code, the responsible methods were made to behave as follows. For the forward kinematics, the code is designed to plug the DH parameters calculated earlier into the appropriate locations in a DH matrix, and then multiply them together to get the overall homogeneous transform. After that, the theta values obtained from the robot model are plugged in, and the resulting location of the end-effector is obtained from the resulting matrix. An example usage of these calculations, as well as the code which generated it, can be seen in Figure D.

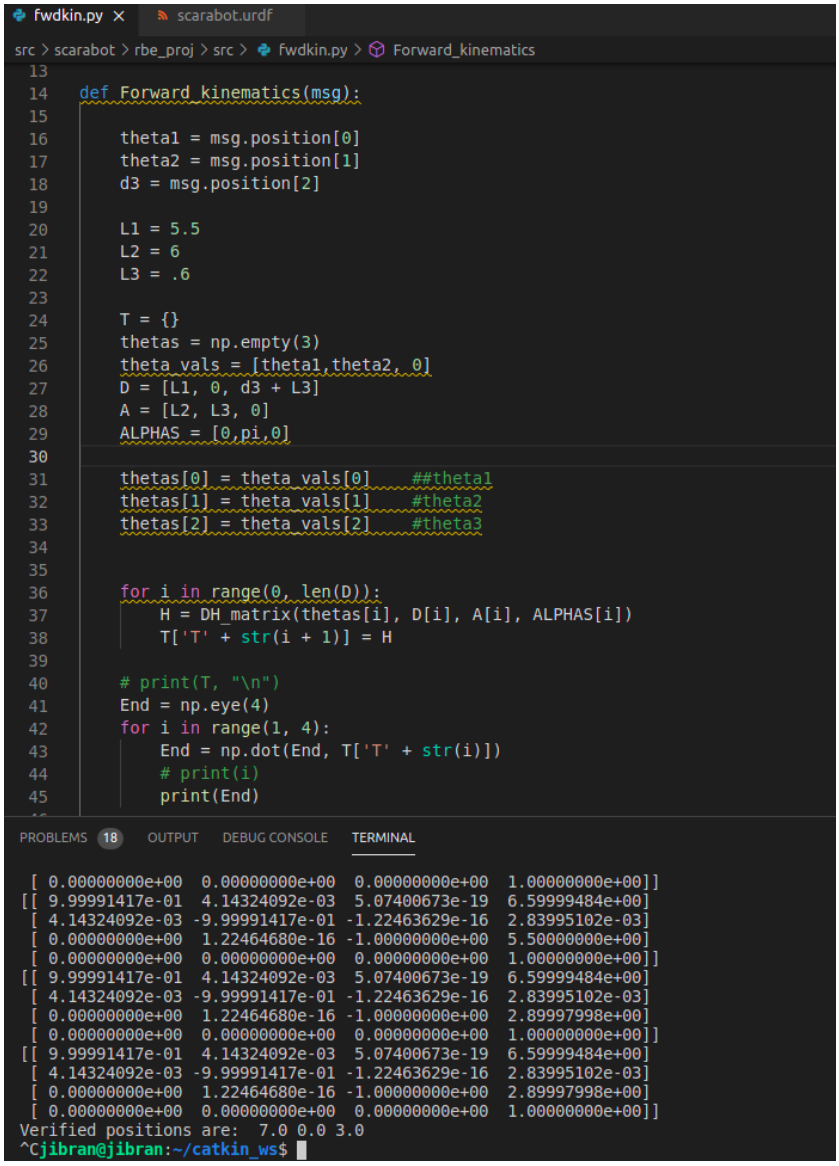

Figure 4: The forwards kinematics code and sample output

(III) INVERSE KINEMATICS

For the inverse kinematics, a server was created which consists of a client, which handles calls to the server, and the server itself, which performs the heavy lifting. The server receives the end effectors position .Those values are then plugged into the derived inverse

kinematic equations, and the joint values are obtained with math derived from the logic used in the two-link example (Figure 5). Also, the server has additional printing functionality in the client, which displays the information put in, as well as the results (Figure 6). Also, if the server is given an invalid number of arguments, it tells the user what the input should be instead, making bugs easier to diagnose. An important factor to note with the z-axis for inverse kinematics is that due to how the control of the prismatic joint is out of the scope of this project, it stays at z=3. This is due to this being set as the lower limit of the joint's movement.

For testing the inverse kinematics, first run the server node. Then you can either (i) run the client node by passing it [x y z] arguments, or alternatively (ii) you can use the following command:

rosservice call /joint_var_calc "xc: 7.0
yc: 0.0
zc: 5.0"

The above values are for home position. Fell free to adjust the values (within the workspace).

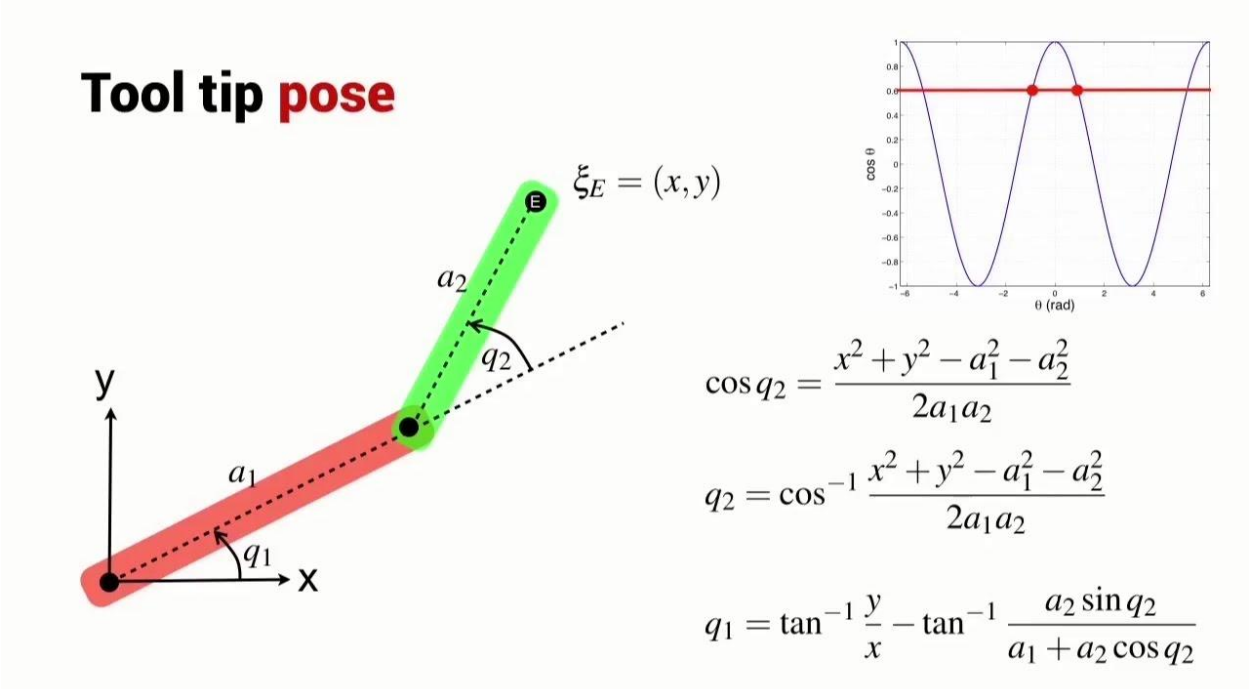

Figure 5: The logic used to derive the inverse kinematics (for the first two joints)

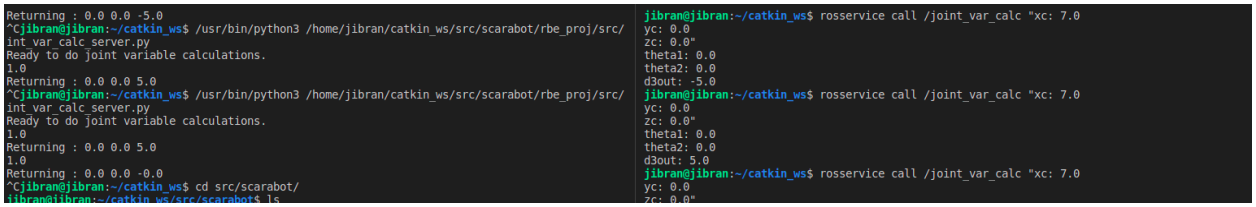

Figure 6: Sample output from the inverse kinematics implementation, calculated for the unpaused gazebo state (as it is the state in which the /joint_states topic is activated).

(IV) GAZEBO ROBOT MODEL

To create the robot model, a combination of the Gazebo model maker and raw urdf code was used. The model maker was used in order to determine the basic geometry of the robot by using the basic provided shapes. However, the model generated by the model maker had issues, such as not being grounded to the world, which made the robot wobble, as well as the fact that the model maker saves models as an sdf file, which cannot be used to obtain the joint data necessary for our joint program to work correctly. In order to connect the robot to the ground properly, a link to the ground was coded in, as well as a base with a large amount of mass relative to the rest of the robot. In order to convert the sdf code into urdf, a conversion tool called sdf2urdf, which was found on github, was used. The structure of the robot is the same regardless of the state, but it was observed that slight visual changes occur between when it is paused and unpaused (Figures 7 and 8).

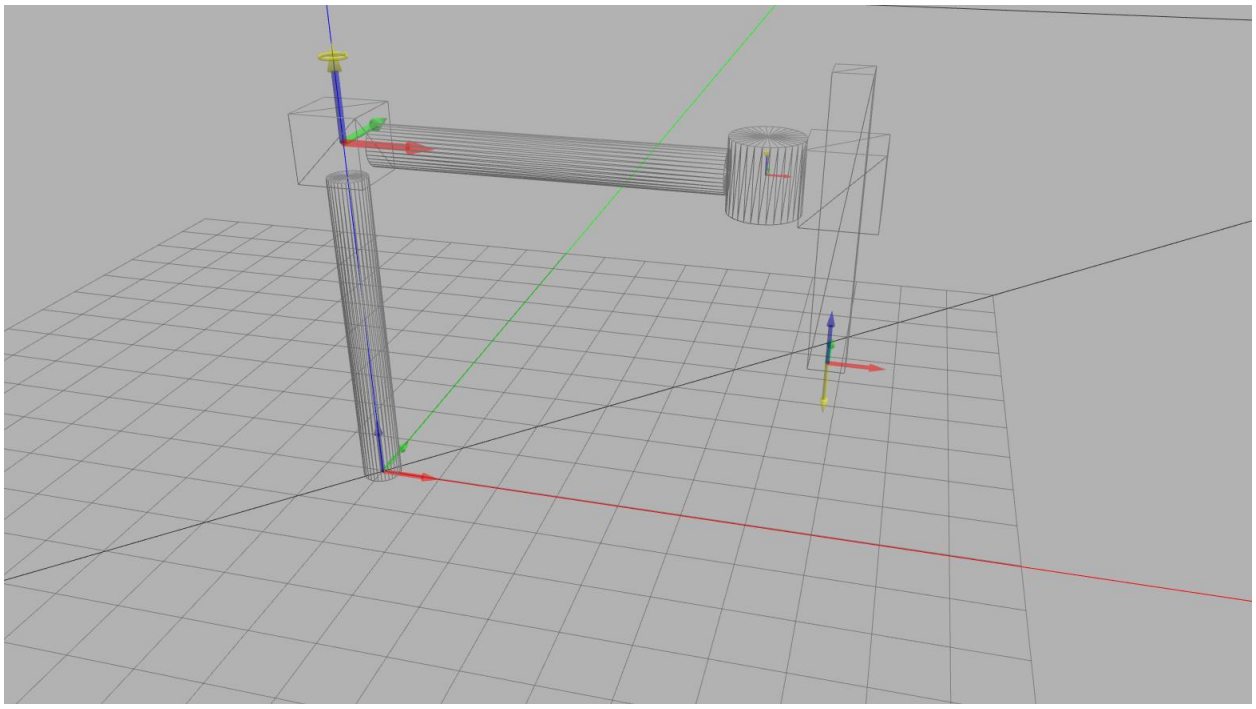

Figure 7: An image of the model in an unpaused state (controllers that freeze the motion of joints, haven't been configured yet, as they are outside the scope of this assignment)

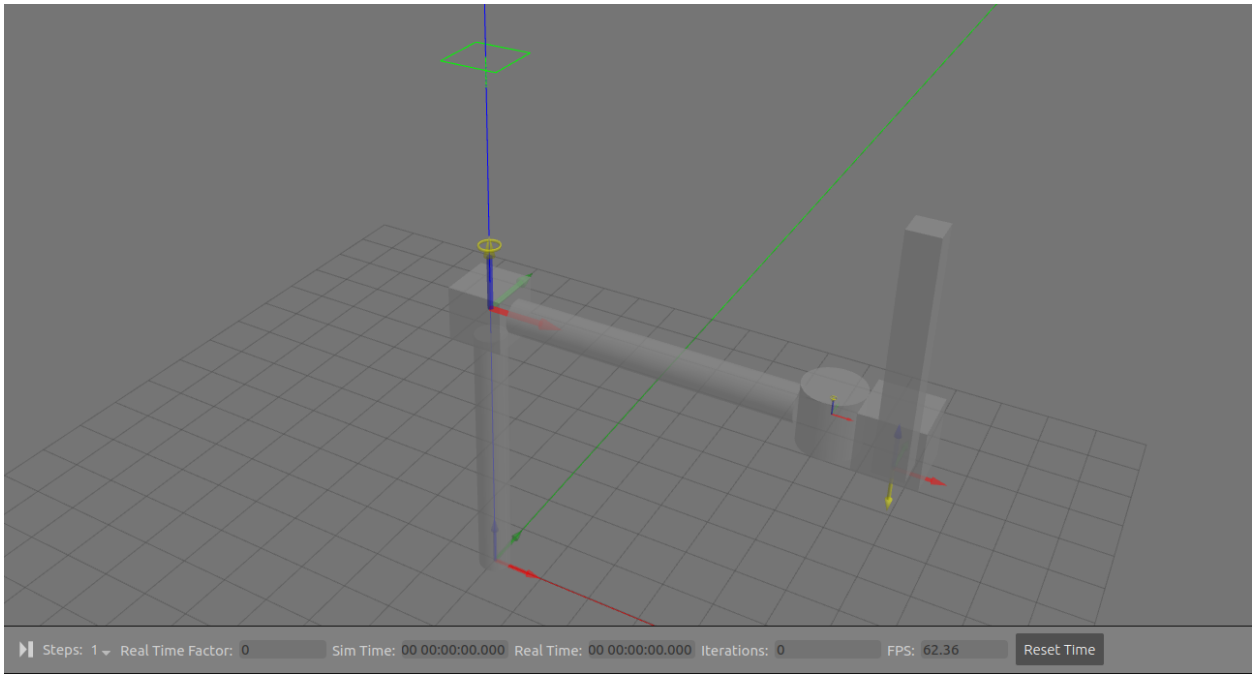

Figure 8: The Robot model in a paused state

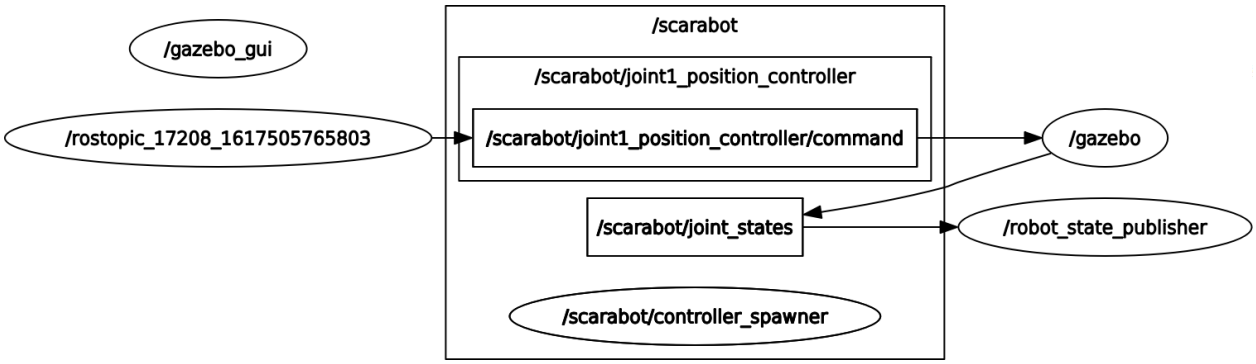

Figure 9: The structure of the nodes utilized in the program and the topics and services that relate them,, as shown by rqt_graph

(V) FILE STRUCTURE:

As seen in Fig. 10, there are 4 packages of concern. All packages are independent ROS packages with their own dependencies but interact with each other in the launch file contained in the rbe_proj directory(arm_world.launch). To launch everything enter the following command:

roslaunch rbe_proj arm_world.launch

The scarabot_description contains the urdf file which describes the robot. A gazebo-Ros plugin is also included in it. It also contains a test launch file to visualize the robot in RViz. The launch file also launches joint_state_publisher. You can play with the joints in RViz by adjusting the parameters in joint_state_publisher gui. The launch file takes the previously stored rviz configuration(scarabot.rviz), feel free to remove that argument.

Further, a scarabot_gazebo directory launches gazebo in a paused state and also loads the urdf into the parameter server and also spawns the controllers from the scarabot_control directory. In the scarabot_control directory. The controllers are defined in the scarabot_control.yaml. These 3 joint controllers are linked to the joints of the robot by using the names of the joints as defined in the urdf file **(joint_2, joint_5 and joint_6**); the parameters of the controllers are arbitrary for now.

Once the launch file (mentioned above) is run, it publishes a topic called **/scarabot/joint_states** instead of the /joint_states as this has been 'remapped' into the launch file(of scarabot_control). These joint_states are received by the forward kinematics node which calculates the end position of robot. The validity of this has been tested as discussed above.

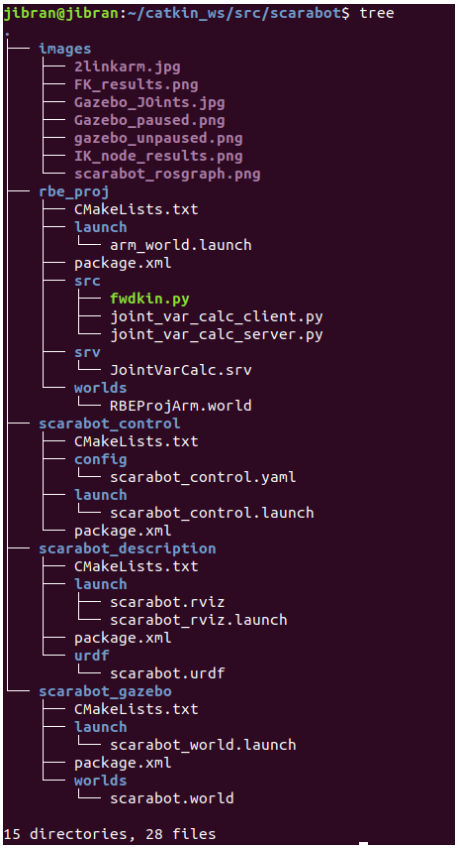

Figure 10: File hierarchy

(VI) **CONCLUSION**

The bot can output its state and is fully customizable, should any changes be needed of lets say, a new world, where the environment has to be changed for a specific task or if the bot' has to be sized up or down. The forward kinematics node is quite flexible as well. The next steps are to configure the controllers, which have been added; but will be brought to working condition in the next project assignment.