Package overview:

- Inside `catkin_ws/src`, the main package is `scara_robot`. It does not directly contain any nodes or launch files, but is a way to organize all of the other nodes.
  - The `scara_gazebo` package includes the launch files for the gazebo world.
  - The `scara_description` package includes the URDF files for the robot as well as the rviz launch files.
  - The `gazebp_publish` package includes the launch file to allow for the joint states to be published from gazebo.
  - The `scara_forward_kinematics` folder is the pub/sub package that subscribes to the joint states, calculates the forward kinematics, and publishes the pose.
  - The `scara_inverse_kinematics` folder is the service/client package that ingests a desired end effector pose and returns the joint position.

1. **Create the robot:**

   - Spawn the robot to the ROS-Gazebo environment. Include and image of the robot.
   - Include the robot definition file.

   Using the URDF file format, we created a `scara.xacro` file which describes the SCARA robot that can be seen in the below figure. This robot is created using six links and five joints; see the tables below for the specific joints/links used and their functions. We additionally use a camera joint and link that is used to center the camera on our SCARA robot. The lengths of the links used in this SCARA robot are completely configurable but for this project arbitrary dimensions were used that allowed the user to observe the operation of the robot easily. Additionally the rotational limits of all of the joints (except for the fixed joints) are completely configurable as well (see the table below for current limits). We further added a `materials.xacro` URDF file to allow multiple colors to be implemented for all of the links so that it's easy to differentiate between them.

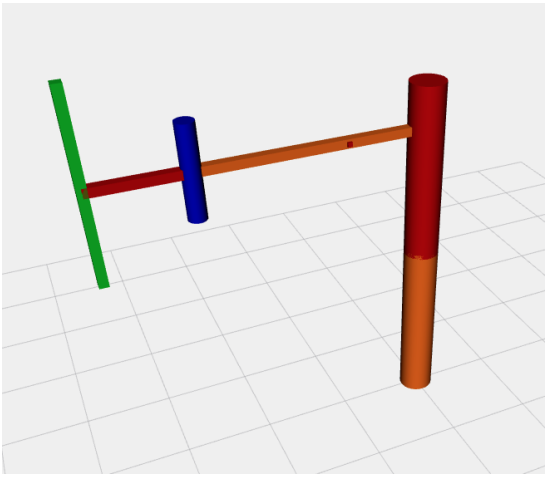


Figure 1: SCARA robot in rviz

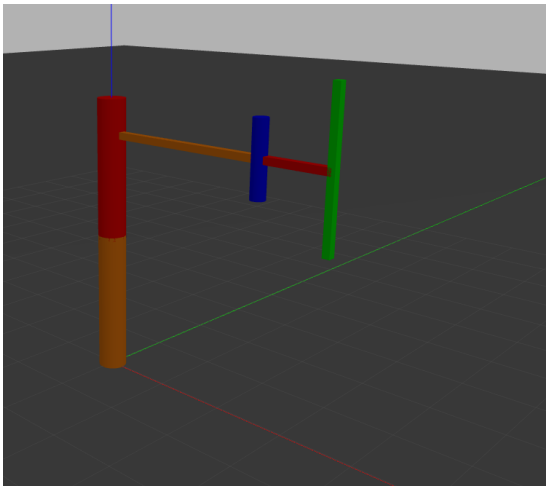


Figure 2: SCARA robot in gazebo

The following joint and link definitions were used:

Joint Name	Joint Type	Joint Limits
Joint 1	Continuous	$-180°$ to $180°$
Joint 2	Fixed	N/A
Joint 3	Revolute	$-90°$ to $90°$
Joint 4	Fixed	N/A
Joint 5	Prismatic	0m to 1m

Table 1: Joints

Link Name	Link Shape	Link Color	Link Purpose
Link 1	Cylinder	Orange	Connects to base frame
Link 2	Cylinder	Red	Models θ_1 Rotation
Link 3	Box	Orange	Models θ_1 Rotation with link 2
Link 4	Cylinder	Blue	Models θ_2 Rotation
Link 5	Box	Red	Models θ_2 Rotation with link 4
Link 6	Box	Green	Translates along d_3

Table 2: Links

To begin the simulation:

- `roslaunch scara_gazebo scara_world.launch`

Once the simulation begins, a force/torque can be applied to the joints to induce rotation/translation, as seen below (link 3 would require a constant force to keep from dropping down):

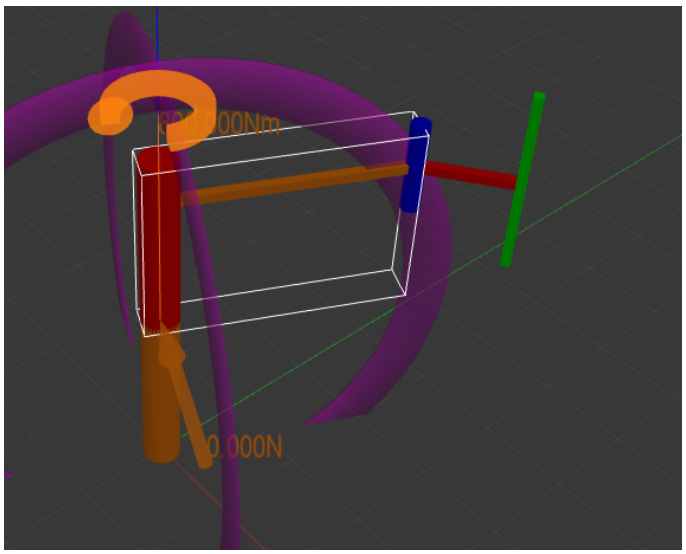


Figure 3: SCARA robot in gazebo after forces/torques applied

The robot description file and gazebo/rviz launch files can be found in the `scara_description` and `scara_gazebo` folders.

2. **Forward Kinematics:** Implement a FK node that

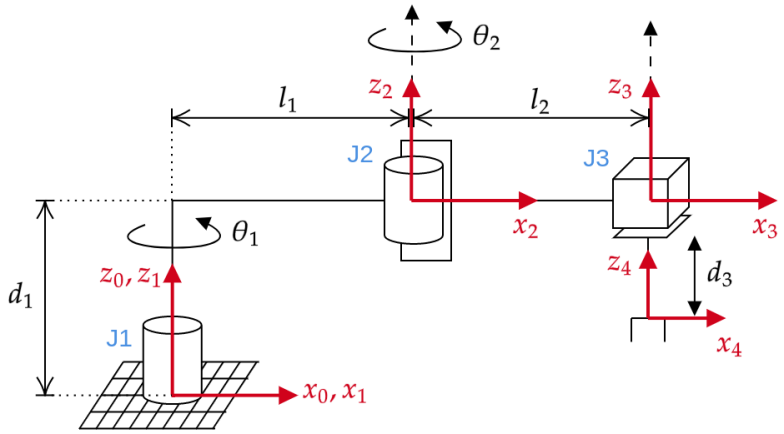
   - Subscribes to the joint values topic and reads them from the Gazebo simulator.
   - Calculates the pose of the end-effector.
   - Publishes the pose as a ROS topic (inside the callback function that reads the joint values).

   Print the resulting pose to the terminal using the `rostopic echo` command and include a screenshot of the results.

   We begin by assigning coordinate frames to the manipulator:

   We can then formulate the DH parameters coordinating to the links:

Link	θ_i	d_i	a_i	α_i
1	θ_1^*	d_1	l_1	0
2	θ_2^*	0	l_2	0
3	0	$-d_3^*$	0	0



Next, we can calculate the transformations for each frame:

$$T_{i+1}^{i} = \text{Rot}_z(\theta_i)\text{Trans}_z(d_i)\text{Trans}_x(a_i)\text{Rot}_x(\alpha_i) = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_2^1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_2^1 = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1\cos 0 & \sin\theta_1\sin 0 & l_1\cos\theta_1 \\ \sin\theta_1 & \cos\theta_1\cos 0 & -\cos\theta_1\sin 0 & l_1\sin\theta_1 \\ 0 & \sin 0 & \cos 0 & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & l_1\cos\theta_1 \\ \sin\theta_1 & \cos\theta_1 & 0 & l_1\sin\theta_1 \\ 0 & 0 & 1 & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_3^2 = \begin{bmatrix} \cos\theta_2 & -\sin\theta_2\cos 0 & \sin\theta_2\sin 0 & l_2\cos\theta_2 \\ \sin\theta_2 & \cos\theta_2\cos 0 & -\cos\theta_2\sin 0 & l_2\sin\theta_2 \\ 0 & \sin 0 & \cos 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & l_2\cos\theta_2 \\ \sin\theta_2 & \cos\theta_2 & 0 & l_2\sin\theta_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_4^3 = \begin{bmatrix} \cos 0 & -\sin 0\cos 0 & \sin 0\sin 0 & 0\cos 0 \\ \sin 0 & \cos 0\cos 0 & -\cos 0\sin 0 & 0\sin 0 \\ 0 & \sin 0 & \cos 0 & -d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The combined transformation of the end effector is:

$$T_4^0 = T_1^0 T_2^1 T_3^2 T_4^3$$

This calculation is used in the forward kinematic function `calc_homogeneous_transform(q)`:

```
def calc_homogeneous_transform(q): # calculate the homogeneous transform from the base frame to EE
        # q = [th1, th2, d3]
        th1 = q[0]
        th2 = q[1]
        d3 = q[2]

        T1_0 = np.matrix([[1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]]) # frame 1 w.r.t 0
        T2_1 = np.matrix([[math.cos(th1),-math.sin(th1),0,l1*math.cos(th1)],[math.sin(th1),math.cos(th1),0,l1*math.sin(th1)],[0,0,1,d1],[0,0,0,1]])      # frame 2 w.r.t 1
```

```
            T3_2 = np.matrix([[math.cos(th2),-math.sin(th2),0,l2*math.cos(th2)],[
                math.sin(th2),math.cos(th2),0,l2*math.sin(th2)],[0,0,1,0],[0,0,0,1]])
                    # frame 3 w.r.t 2
            T4_3 = np.matrix([[1,0,0,0],[0,1,0,0],[0,0,1,-d3],[0,0,0,1]]) # frame 4
                w.r.t 3

            T_EE = T1_0.dot(T2_1).dot(T3_2).dot(T4_3)

            return T_EE
```

To run and test the subscriber/publisher function:

- Run `roslaunch scara_gazebo scara_world.launch`
- In a new window `roslaunch gazebo_publish gazebo_publish.launch` which allows the joint states to be published.
- In a new window `rostopic list` should show /scara/joint_states
- `rostopic echo /scara/joint_states` should show the joint states printing
- `rosrun scara_forward_kinematics configuration_to_operational_sub.py` will run the program that subscribes to the joint states, calculates the forward kinematics, and publishes the end-effector pose back to the `Pose` topic. The print out can be seen below:

```
Received joint parameters: [0.000007, 0.000003, 0.000001] (th1,th2,d3) (radians, meters)

Converted end-effector pose: p = [3.000000, 0.000025, 3.499999, 0.000000, 0.000000, 0.000010] (x,y,z,phi,theta,
psi) (ZYZ) (meters, radians)
```

- We can also see the `Pose` topic getting published by running `rostopic echo /scara/pose`

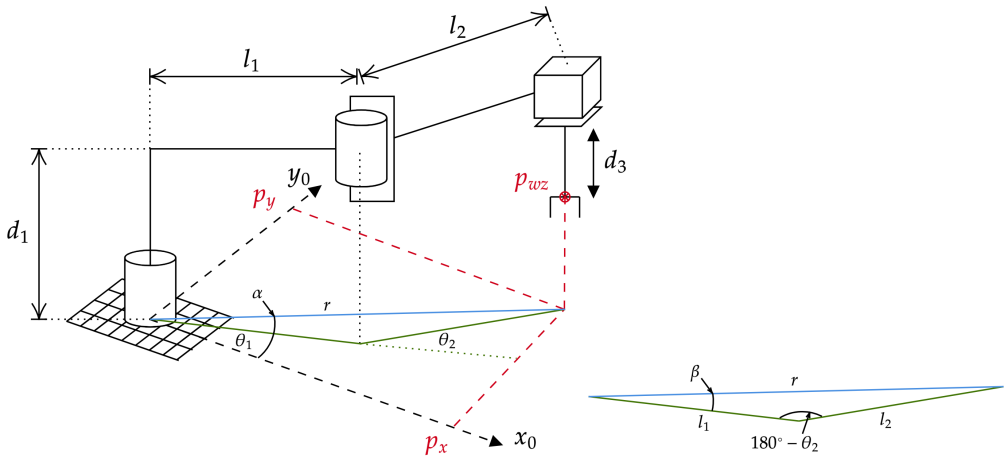
```
---
x: 2.99999999989
y: 2.50742131251e-05
z: 3.49999944466
phi: 0.0
theta: 0.0
psi: 1.0119293119e-05
---
```

Note: the above shows the robot in the home position (i.e on the $x-z$ plane)

3. **Inverse Kinematics:**

- Implement an IK node (separate node) that has a service client that take a desired pose of the end effector and returns joint positions as a response.
- Test your node with `rosservice call`. Include a screenshot of the results.

The following definitions can be used to calculate the inverse kinematics:



The large right triangle can be used to calculate the following:

$$r = \sqrt{p_x^2 + p_y^2}$$
$$A = \frac{p_x}{r} = \cos\alpha \Rightarrow \sin\alpha = \pm\sqrt{1 - A^2} \Rightarrow \alpha = \text{atan2}\left(\pm\sqrt{1 - A}, A\right)$$

The law of cosines can be used on the other triangle to calculate β and θ_1:

$$l_2^2 = r^2 + l_1^2 - 2rl_1\cos\beta$$
$$\cos\beta = \frac{r^2 + l_1^2 - l_2^2}{2rl_1} = C \Rightarrow \sin\beta = \pm\sqrt{1 - C^2}$$
$$\beta = \text{atan2}\left(\pm\sqrt{1 - C^2}, C\right)$$
$$\theta_1 = \alpha - \beta$$

The law of cosines can be used again to calculate θ_1:

$$r^2 = l_1^2 + l_2^2 - 2l_1l_2\cos(180 - \theta_2)$$
$$r^2 = l_1^2 + l_2^2 + 2l_1l_2\cos(\theta_2)$$
$$\cos\theta_2 = \frac{r^2 - l_1^2 - l_2^2}{2l_1l_2} = D \Rightarrow \sin\theta_2 = \pm\sqrt{1 - D^2}$$
$$\theta_2 = \text{atan2}\left(\pm\sqrt{1 - D^2}, D\right)$$

Lastly, $d3$ is calculated simply as follows:

$$d_3 = d_1 - p_z$$

The above equations are included in the server `inverse_server.cpp`:

```
double D = -  (((l1*l1)+(l2*l2)-(x*x+y*y))/(2*l1*l2));
double C =  (((l1*l1)+x*x+y*y-(l2*l2))/(2*l1*sqrt(x*x+y*y)));

double B = sqrt(1-D*D);
double E = sqrt(1-C*C);

double F = x/sqrt(x*x+y*y);
double G = sqrt(1-F*F);
double alpha = atan2(G, F);

res.theta1 = alpha-atan2(E, C);
res.theta2 = atan2(B, D);
res.d3 = d1 - z;
```

To test the node:

- `rosrun scara_inverse_kinematics inverse_server`
- `rosrun scara_inverse_kinematics inverse_client X Y Z`

```
chris@ubuntu-desktop:~/rbe500_team2_pa1/catkin_ws$ rosrun scara_inverse_kinematics inverse_client 3 0 3.5
```

- The server will then print out the incoming message and send back the response. The following shows the home position (i.e. all joints at 0):

```
^Cchris@ubuntu-desktop:~/rbe500_team2_pa1/catkin_ws$ rosrun scara_inverse_kinematics inverse_server
[ INFO] [1625010628.806978197]: Ready to find Joints.
[ INFO] [1625010642.827262111]: request: x=3.000000, y=0.000000, z=3.500000
[ INFO] [1625010642.827302680]: sending back response: theta1=0.000000, theta2=0.000000, d3=0.000000
```

- the server can also be tested using `rosservice`:

```
chris@ubuntu-desktop:~/rbe500_team2_pa1/catkin_ws$ rosservice call /inverse_server "x: 2.2
y: 0.2
z: 3.4"
```

which results in the following:

```
^Cchris@ubuntu-desktop:~/rbe500_team2_pa1/catkin_ws$ rosrun scara_inverse_kinematics inverse_server
[ INFO] [1625011012.499403332]: Ready to find Joints.
[ INFO] [1625011017.512260601]: request: x=2.200000, y=0.200000, z=3.400000
[ INFO] [1625011017.512303646]: sending back response: theta1=-0.378879, theta2=1.600801, d3=0.100000
```