# 0. Pub/Sub to publish and receive name in ROS

1. **AIM**: To print the name via pub/sub
2. **METHOD**:

* Create catkin workspace: mkdir -p catkin\_ws/src
* Create catkin package: catkin\_create\_pkg exp0\_postlab rospy roscpp std\_msgs
* Create scripts: cd src/exp0\_postlab/src && mkdir scripts && cd scripts && touch publisher.py && touch subscriber.py
* Add the scripts in this directory
* Catkin Make: catkin\_make

1. **CODE**:

* publisher.py

#!/usr/bin/env python  
import rospy  
from std\_msgs.msg import String  
  
def publisher():  
 pub = rospy.Publisher('chatter', String, queue\_size=10)  
 rospy.init\_node('talker', anonymous=True)  
 rate = rospy.Rate(1) # 10hz  
 while not rospy.is\_shutdown():  
 name = "Aaryamann Challani"  
 rospy.loginfo(name)  
 pub.publish(name)  
  
 rate.sleep()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 try:  
 publisher()  
 except rospy.ROSInterruptException:  
 pass

* subscriber.py

#!/usr/bin/env python  
import rospy  
from std\_msgs.msg import String  
  
def callback(data):  
 rospy.loginfo(rospy.get\_caller\_id() + "Name:%s", data.data)  
  
def subscriber():  
 rospy.init\_node('listener', anonymous=True)  
 rospy.Subscriber('chatter', String, callback)  
 rospy.spin()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 subscriber()

1. **RESULT**: Successfully published and received messages via ROS’s pubsub transport

# 1. Move the turtle bot in D shape

1. **AIM**: Move the turtlebot in a D path
2. **METHOD**:

* Create catkin package: catkin\_create\_pkg exp1\_postlab rospy roscpp std\_msgs
* Create script: cd src/exp1\_postlab/src && mkdir scripts && cd scripts && touch turtlesim\_d.py
* Add the script in this directory
* Catkin Make: catkin\_make
* Start roscore: roscore
* Start the turtlesim: rosrun turtlesim turtlesim\_node
* Run the script: rosrun exp1\_postlab turtlesim\_d.py

1. **CODE**:

* turtlesim\_d.py:

#!/usr/bin/env python3  
  
import rospy  
from geometry\_msgs.msg import Twist  
from turtlesim.msg import Pose  
import math  
import time  
from std\_srvs.srv import Empty  
  
X = 0.0  
Y = 0.0  
yaw = 0.0  
  
  
def pose\_callback(pose):  
 global X, Y, yaw  
 rospy.loginfo("X=%f, Y=%f\n", pose.x, pose.y)  
 X = pose.x  
 Y = pose.y  
 yaw = pose.theta  
  
  
def move(speed, distance, is\_forward):  
 velocity\_message = Twist()  
  
 global X, Y  
 X0 = X  
 Y0 = Y  
  
 if is\_forward:  
 velocity\_message.linear.x = abs(speed)  
 else:  
 velocity\_message.linear.x = -abs(speed)  
  
 distance\_moved = 0.0  
 loop\_rate = rospy.Rate(10)  
 cmd\_vel\_topic = '/turtle1/cmd\_vel'  
 velocity\_publisher = rospy.Publisher(cmd\_vel\_topic, Twist, queue\_size=10)  
  
 while True:  
 rospy.loginfo("Turtlesim moves forward")  
 velocity\_publisher.publish(velocity\_message)  
 loop\_rate.sleep()  
  
 # rospy.loginfo("%f %f %f %f", X,Y,X0,Y0)  
  
 distance\_moved = abs(0.5 \* math.sqrt(((X - X0) \*\* 2) + ((X - Y0) \*\* 2)))  
 print(distance\_moved)  
  
 if not (distance\_moved < distance):  
 rospy.loginfo("reached")  
 rospy.logwarn("Stopping the Robot")  
 break  
  
 velocity\_message.linear.x = 0  
 velocity\_publisher.publish(velocity\_message)  
  
  
def rotate(angular\_speed\_degree, relative\_angle\_degree, clockwise):  
 global yaw  
  
 velocity\_message = Twist()  
 velocity\_message.linear.x = 0  
 velocity\_message.angular.z = 0  
  
 theta0 = yaw  
 angular\_speed = math.radians(abs(angular\_speed\_degree))  
  
 if clockwise:  
 velocity\_message.angular.z = -abs(angular\_speed)  
  
 else:  
 velocity\_message.angular.z = abs(angular\_speed)  
  
 angle\_moved = 0.0  
 loop\_rate = rospy.Rate(10)  
 and\_vel\_topic = '/turtle1/cmd\_vel'  
  
 velocity\_publisher = rospy.Publisher(cmd\_vel\_topic, Twist, queue\_size=10)  
 t0 = rospy.Time.now().to\_sec()  
  
 while (True):  
  
 rospy.loginfo("Turtlesim rotates")  
 velocity\_publisher.publish(velocity\_message)  
  
 t1 = rospy.Time.now().to\_sec()  
 current\_angle\_degree = (t1 - t0) \* angular\_speed\_degree  
 loop\_rate.sleep()  
  
 if current\_angle\_degree > relative\_angle\_degree:  
 rospy.loginfo("reached")  
 break  
  
 velocity\_message.angular.z = 0  
 velocity\_publisher.publish(velocity\_message)  
  
  
def go\_to\_goal(x\_goal, y\_goal):  
 global X  
 global Y, yaw  
  
 velocity\_message = Twist()  
 cmd\_vel\_topic = '/turtle1/cmd\_vel'  
  
 while True:  
 K\_linear = 0.5  
  
 distance = abs(math.sqrt(((x\_goal - X) \*\* 2) + ((y\_goal - Y) \*\* 2)))  
 linear\_speed = distance \* K\_linear  
 K\_angular = 4.0  
  
 desired\_angle\_goal = math.atan2(y\_goal - Y, x\_goal - X)  
 angular\_speed = (desired\_angle\_goal - yaw) \* K\_angular  
  
 velocity\_message.linear.x = linear\_speed  
 velocity\_message.angular.z = angular\_speed  
 velocity\_publisher.publish(velocity\_message)  
  
 print('x=', X, 'y=', Y)  
  
 if distance < 0.01:  
 break  
  
  
def setDesiredOrientation(desired\_angle\_radians):  
 relative\_angle\_radians = desired\_angle\_radians - yaw  
  
 if relative\_angle\_radians < 0:  
 clockwise = 1  
 else:  
 clockwise = 0  
  
 print(relative\_angle\_radians)  
 print(desired\_angle\_radians)  
 rotate(30, math.degrees(abs(relative\_angle\_radians)), clockwise)  
  
  
def move\_turtlebot\_d(speed):  
 global X, Y, yaw  
 velocity\_message = Twist()  
 velocity\_message.linear.x = speed  
 velocity\_message.angular.z = 2.5  
  
 while True:  
 velocity\_publisher.publish(velocity\_message)  
  
 if 5.44 < X < 5.54:  
 break  
  
 velocity\_message = Twist()  
 velocity\_publisher.publish(velocity\_message)  
  
 # Orient to Spawn  
 setDesiredOrientation(-1.5)  
  
 # Reach Spawn  
 curr\_dist = Y  
 go\_to\_goal(5.5,5.5)  
  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
 try:  
 rospy.init\_node('turtlesim\_motion\_pose', anonymous=True)  
 cmd\_vel\_topic = '/turtle1/cmd\_vel'  
  
 velocity\_publisher = rospy.Publisher(cmd\_vel\_topic, Twist, queue\_size=10)  
 position\_topic = '/turtle1/pose'  
  
 rospy.Subscriber(position\_topic, Pose, pose\_callback)  
  
 move\_turtlebot\_d(6)  
  
 except rospy.ROSInterruptException:  
 rospy.loginfo("node terminated")

1. **RESULT**: Successfully launched a turtlebot and moved it in the D shape

# 2. Move the turtle bot in Hexagon shape

1. **AIM**: Move the turtlebot in a Hexagon path
2. **METHOD**:

* Create catkin package: catkin\_create\_pkg exp2\_postlab rospy roscpp std\_msgs
* Create script: cd src/exp2\_postlab/src && mkdir scripts && cd scripts && touch turtlesim\_hex.py
* Add the script in this directory
* Catkin Make: catkin\_make
* Start roscore: roscore
* Start the turtlesim: rosrun turtlesim turtlesim\_node
* Run the script: rosrun exp2\_postlab turtlesim\_hex.py

1. **CODE**:

* turtlesim\_hex.py

#!/usr/bin/env python3  
  
import rospy  
from geometry\_msgs.msg import Twist  
from turtlesim.msg import Pose  
import math  
import time  
from std\_srvs.srv import Empty  
  
X = 0.0  
Y = 0.0  
yaw = 0.0  
  
  
def pose\_callback(pose):  
 global X, Y, yaw  
 rospy.loginfo("X=%f, Y=%f\n", pose.x, pose.y)  
 X = pose.x  
 Y = pose.y  
 yaw = pose.theta  
  
  
def move(speed, distance, is\_forward):  
 velocity\_message = Twist()  
  
 global X, Y  
 X0 = X  
 Y0 = Y  
  
 if is\_forward:  
 velocity\_message.linear.x = abs(speed)  
 else:  
 velocity\_message.linear.x = -abs(speed)  
  
 distance\_moved = 0.0  
 loop\_rate = rospy.Rate(10)  
 cmd\_vel\_topic = '/turtle1/cmd\_vel'  
 velocity\_publisher = rospy.Publisher(cmd\_vel\_topic, Twist, queue\_size=10)  
  
 while True:  
 rospy.loginfo("Turtlesim moves forward")  
 velocity\_publisher.publish(velocity\_message)  
 loop\_rate.sleep()  
  
 # rospy.loginfo("%f %f %f %f", X,Y,X0,Y0)  
  
 distance\_moved = math.sqrt(((X - X0) \*\* 2) + ((Y - Y0) \*\* 2))  
 print(distance\_moved,X,Y,X0,Y0)  
  
 if not (distance\_moved < distance):  
 rospy.loginfo("reached")  
 rospy.logwarn("Stopping the Robot")  
 break  
  
 velocity\_message.linear.x = 0  
 velocity\_publisher.publish(velocity\_message)  
  
  
def rotate(angular\_speed\_degree, relative\_angle\_degree, clockwise):  
 global yaw  
  
 velocity\_message = Twist()  
 velocity\_message.linear.x = 0  
 velocity\_message.angular.z = 0  
  
 theta0 = yaw  
 angular\_speed = math.radians(abs(angular\_speed\_degree))  
  
 if clockwise:  
 velocity\_message.angular.z = -abs(angular\_speed)  
  
 else:  
 velocity\_message.angular.z = abs(angular\_speed)  
  
 angle\_moved = 0.0  
 loop\_rate = rospy.Rate(10)  
 and\_vel\_topic = '/turtle1/cmd\_vel'  
  
 velocity\_publisher = rospy.Publisher(cmd\_vel\_topic, Twist, queue\_size=10)  
 t0 = rospy.Time.now().to\_sec()  
  
 while (True):  
  
 rospy.loginfo("Turtlesim rotates")  
 velocity\_publisher.publish(velocity\_message)  
  
 t1 = rospy.Time.now().to\_sec()  
 current\_angle\_degree = (t1 - t0) \* angular\_speed\_degree  
 loop\_rate.sleep()  
  
 if current\_angle\_degree > relative\_angle\_degree:  
 rospy.loginfo("reached")  
 break  
  
 velocity\_message.angular.z = 0  
 velocity\_publisher.publish(velocity\_message)  
  
  
def go\_to\_goal(x\_goal, y\_goal):  
 global X  
 global Y, yaw  
  
 velocity\_message = Twist()  
 cmd\_vel\_topic = '/turtle1/cmd\_vel'  
  
 while True:  
 K\_linear = 0.5  
  
 distance = abs(math.sqrt(((x\_goal - X) \*\* 2) + ((y\_goal - Y) \*\* 2)))  
 linear\_speed = distance \* K\_linear  
 K\_angular = 4.0  
  
 desired\_angle\_goal = math.atan2(y\_goal - Y, x\_goal - X)  
 angular\_speed = (desired\_angle\_goal - yaw) \* K\_angular  
  
 velocity\_message.linear.x = linear\_speed  
 velocity\_message.angular.z = angular\_speed  
 velocity\_publisher.publish(velocity\_message)  
  
 print('x=', X, 'y=', Y)  
  
 if distance < 0.01:  
 break  
  
  
def setDesiredOrientation(desired\_angle\_radians):  
 relative\_angle\_radians = desired\_angle\_radians - yaw  
  
 if relative\_angle\_radians < 0:  
 clockwise = 1  
 else:  
 clockwise = 0  
  
 print(relative\_angle\_radians)  
 print(desired\_angle\_radians)  
 rotate(30, math.degrees(abs(relative\_angle\_radians)), clockwise)  
  
  
def hexagon(side\_length):  
 for i in range(6):  
 print("Moving on {} edge".format(i))  
 move(1.0,3.0,True)  
 rotate(10,60,False)  
  
  
  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
 try:  
 rospy.init\_node('turtlesim\_motion\_pose', anonymous=True)  
 cmd\_vel\_topic = '/turtle1/cmd\_vel'  
  
 velocity\_publisher = rospy.Publisher(cmd\_vel\_topic, Twist, queue\_size=10)  
 position\_topic = '/turtle1/pose'  
  
 rospy.Subscriber(position\_topic, Pose, pose\_callback)  
 time.sleep(1)  
  
 hexagon(3.0)  
  
 except rospy.ROSInterruptException:  
 rospy.loginfo("node terminated")

1. **RESULT**: Successfully launched a turtlebot and moved it in the hexagon shape

# 3. 2-wheeled robot with caster wheel

1. **AIM**: To create a 2-wheeled robot with caster wheel using URDF
2. **METHOD**:

* Create catkin package: catkin\_create\_pkg exp3\_postlab
* Create urdf files: cd src/exp3\_postlab/src && mkdir urdf && mkdir launch && cd urdf && touch bot.urdf && cd ../launch && touch gazebo.launch
* Add the launch file and urdf definition in this directory
* Launch using roslaunch exp3\_postlab gazebo.launch

1. **CODE**

* bot.urdf:

<?xml version='1.0'?>  
<robot name="exp3">  
  
 <gazebo>  
 <static>False</static>  
 </gazebo>  
  
  
 <link name="dummy"/>  
  
 <!-- Base Link -->  
 <link name="base\_link">  
 <visual>  
 <material name="red">  
 <color rgba="1 0 0 1"/>  
 </material>  
 <origin xyz="0 0 0" rpy="0 0 0" />  
 <geometry>  
 <box size="0.7 0.7 0.30"/>  
 </geometry>  
 </visual>  
 <!-- Base collision, mass and inertia -->  
 <collision>  
 <origin xyz="0 0 0" rpy="0 0 0" />  
 <geometry>  
 <box size="0.7 0.7 0.30"/>  
 </geometry>  
 </collision>  
 <inertial>  
 <mass value="7"/>  
 <inertia ixx="0.15" ixy="0.0" ixz="0.0" iyy="0.25" iyz="0.0" izz="0.15"/>  
 </inertial>  
  
 <!-- Caster Wheel -->  
 <visual name="caster">  
 <material name="black">  
 <color rgba="0 0 0 1"/>  
 </material>  
 <origin xyz="0.4 0 -0.125" rpy="0 0 0" />  
 <geometry>  
 <sphere radius="0.10" />  
 </geometry>  
 </visual>  
 <!-- Caster collision, mass and inertia -->  
 <collision>  
 <origin xyz="0.4 0 -0.125" rpy="0 0 0" />  
 <geometry>  
 <sphere radius="0.10" />  
 </geometry>  
 </collision>  
 <inertial>  
 <mass value="0.5"/>  
 <inertia ixx="0.0001" ixy="0.0" ixz="0.0" iyy="0.0001" iyz="0.0" izz="0.0001"/>  
 </inertial>  
  
 </link>  
  
 <gazebo reference="base\_link">  
 <material>Gazebo/RedGlow</material>  
 <pose>0 0 3 0 0 0</pose>  
 </gazebo>  
  
 <joint name="dummy\_joint" type="fixed">  
 <parent link="dummy"/>  
 <child link="base\_link"/>  
 </joint>  
  
 <!-- Right Wheel -->  
 <link name="right\_wheel">  
 <visual>  
 <material name="green">  
 <color rgba="0 1 0 1"/>  
 </material>  
 <origin xyz="0 0 0" rpy="1.570795 0 0" />  
 <geometry>  
 <cylinder length="0.1" radius="0.2" />  
 </geometry>  
 </visual>  
 <!-- Right Wheel collision, mass and inertia -->  
 <collision>  
 <origin xyz="0 0 0" rpy="1.570795 0 0" />  
 <geometry>  
 <cylinder length="0.1" radius="0.2" />  
 </geometry>  
 </collision>  
 <inertial>  
 <mass value="0.5"/>  
 <inertia ixx="0.01" ixy="0.0" ixz="0.0" iyy="0.005" iyz="0.0" izz="0.005"/>  
 </inertial>  
  
 </link>  
  
 <gazebo reference="right\_wheel">  
 <material>Gazebo/RedGlow</material>  
 </gazebo>  
  
 <!-- Right Wheel joint -->  
 <joint name="joint\_right\_wheel" type="continuous">  
 <parent link="base\_link"/>  
 <child link="right\_wheel"/>  
 <origin xyz="0 -0.30 0.025" rpy="0 0 0" />   
 <axis xyz="0 1 0" />  
 </joint>  
  
 <!-- Left Wheel -->  
 <link name="left\_wheel">  
 <visual>  
 <material name="blue">  
 <color rgba="0.0 0.0 1 1"/>  
 </material>  
 <origin xyz="0 0 0" rpy="1.570795 0 0" />  
 <geometry>  
 <cylinder length="0.1" radius="0.2" />  
 </geometry>  
 </visual>  
 <!-- Left Wheel collision, mass and inertia -->  
 <collision>  
 <origin xyz="0 0 0" rpy="1.570795 0 0" />  
 <geometry>  
 <cylinder length="0.1" radius="0.2" />  
 </geometry>  
 </collision>  
 <inertial>  
 <mass value="0.5"/>  
 <inertia ixx="0.01" ixy="0.0" ixz="0.0" iyy="0.005" iyz="0.0" izz="0.005"/>  
 </inertial>  
  
 </link>  
  
 <gazebo reference="left\_wheel">  
 <material>Gazebo/Blue</material>  
 </gazebo>  
  
 <!-- Left Wheel joint -->  
 <joint name="joint\_left\_wheel" type="continuous">  
 <parent link="base\_link"/>  
 <child link="left\_wheel"/>  
 <origin xyz="0 0.30 0.025" rpy="0 0 0" />   
 <axis xyz="0 1 0"/>  
 </joint>  
  
</robot>

* gazebo.launch:

<?xml version="1.0" encoding="UTF-8"?>  
<launch>  
   
 <include file="$(find gazebo\_ros)/launch/empty\_world.launch">   
 <arg name="paused" default="false"/>  
 <arg name="use\_sim\_time" default="true"/>  
 <arg name="gui" default="true"/>  
 <arg name="headless" default="false"/>  
 <arg name="debug" default="false"/>  
 </include>  
 <node name="spawn\_urdf" pkg="gazebo\_ros" type="spawn\_model" output="screen"  
 args="-file $(find exp3\_postlab)/urdf/bot.urdf -urdf -z -2.82 -model exp3" />  
  
</launch>

1. **RESULT**: Successfully simulated/defined parameters for a 2 wheeled robot with a caster wheel via URDF/ROS. Observed output in Gazebo.

# 4. 3 Joint Manipulator with gripper as end effector

1. **AIM**: To Create a manipulator using URDF
2. **METHOD**:

* Create catkin package: catkin\_create\_pkg exp4\_postlab
* Create urdf/launch files: cd src/exp4\_postlab/src && mkdir urdf && mkdir launch && cd urdf && touch manipulator.urdf && cd ../launch/ && touch gazebo.launch
* Add/Copy the files from this directory
* Catkin Make: catkin\_make
* Launch the bot: roslaunch exp4\_postlab gazebo.launch
* Note: From ROS Textbook

1. **CODE**:

* rrbot.xacro:

<?xml version="1.0"?>  
<!-- Revolute-Revolute Manipulator -->  
<robot name="rrbot" xmlns:xacro="http://www.ros.org/wiki/xacro">  
  
 <!-- Constants for robot dimensions -->  
 <xacro:property name="PI" value="3.14"/>  
 <xacro:property name="width" value="0.1" /> <!-- Beams are square in length and width -->  
 <xacro:property name="height1" value="2" /> <!-- Link 1 -->  
 <xacro:property name="height2" value="1" /> <!-- Link 2 -->  
 <xacro:property name="height3" value="1" /> <!-- Link 3 -->  
 <xacro:property name="axle\_offset" value="0.05" /> <!-- Space between joint and end of beam -->  
 <xacro:property name="damp" value="0.7" /> <!-- damping coefficient -->  
  
 <!-- Default Inertial -->  
 <xacro:macro name="default\_inertial" params="z\_value i\_value mass">  
 <inertial>  
 <origin xyz="0 0 ${z\_value}" rpy="0 0 0"/>  
 <mass value="${mass}" />  
 <inertia ixx="${i\_value}" ixy="0.0" ixz="0.0"  
 iyy="${i\_value}" iyz="0.0"  
 izz="${i\_value}" />  
 </inertial>  
 </xacro:macro>  
  
 <!-- Import Rviz colors -->  
 <xacro:include filename="$(find manipulator)/urdf/materials.xacro" />  
  
 <!-- Import gripper URDF -->  
 <xacro:include filename="$(find manipulator)/urdf/gripper.xacro" />   
  
 <!-- Import Gazebo elements, including Gazebo colors -->  
 <xacro:include filename="$(find manipulator)/urdf/rrbot.gazebo" />  
  
 <!-- Used for fixing rrbot frame to Gazebo world frame -->  
 <link name="world"/>  
  
 <joint name="fixed" type="fixed">  
 <parent link="world"/>  
 <child link="base\_link"/>  
 </joint>  
  
 <!-- Base Link -->  
 <link name="base\_link">  
 <visual>  
 <origin xyz="0 0 ${height1/2}" rpy="0 0 0"/>  
 <geometry>  
 <box size="${width} ${width} ${height1}"/>  
 </geometry>  
 <material name="red"/>  
 </visual>  
  
 <collision>  
 <origin xyz="0 0 ${height1/2}" rpy="0 0 0"/>  
 <geometry>  
 <box size="${width} ${width} ${height1}"/>  
 </geometry>  
 </collision>  
  
 <xacro:default\_inertial z\_value="${height1/2}" i\_value="1.0" mass="1"/>  
 </link>  
  
 <!-- Joint between Base Link and Middle Link -->  
 <joint name="joint\_base\_mid" type="revolute">  
 <parent link="base\_link"/>  
 <child link="mid\_link"/>  
 <origin xyz="0 ${width} ${height1 - axle\_offset}" rpy="0 0 0"/>  
 <axis xyz="0 1 0"/>  
 <dynamics damping="${damp}"/>  
 <limit effort="100.0" velocity="0.5" lower="-3.14" upper="3.14" />  
 </joint>  
  
 <!-- Middle Link -->  
 <link name="mid\_link">  
 <visual>  
 <origin xyz="0 0 ${height2/2 - axle\_offset}" rpy="0 0 0"/>  
 <geometry>  
 <box size="${width} ${width} ${height2}"/>  
 </geometry>  
 <material name="green"/>  
 </visual>  
  
 <collision>  
 <origin xyz="0 0 ${height2/2 - axle\_offset}" rpy="0 0 0"/>  
 <geometry>  
 <box size="${width} ${width} ${height2}"/>  
 </geometry>  
 </collision>  
  
 <xacro:default\_inertial z\_value="${height2/2 - axle\_offset}" i\_value="1.0" mass="1"/>  
 </link>  
  
 <!-- Joint between Middle Link and Top Link -->  
 <joint name="joint\_mid\_top" type="revolute">   
 <parent link="mid\_link"/>  
 <child link="top\_link"/>  
 <origin xyz="0 ${width} ${height2 - axle\_offset\*2}" rpy="0 0 0"/>  
 <axis xyz="0 1 0"/>   
 <dynamics damping="${damp}"/>   
 <limit effort="100.0" velocity="0.5" lower="-3.14" upper="3.14" />  
 </joint>  
  
 <!-- Top Link -->  
 <link name="top\_link">  
 <visual>  
 <origin xyz="0 0 ${height3/2 - axle\_offset}" rpy="0 0 0"/>  
 <geometry>  
 <box size="${width} ${width} ${height3}"/>  
 </geometry>  
 <material name="blue"/>  
 </visual>  
  
 <collision>  
 <origin xyz="0 0 ${height3/2 - axle\_offset}" rpy="0 0 0"/>  
 <geometry>  
 <box size="${width} ${width} ${height3}"/>  
 </geometry>  
 </collision>  
  
 <xacro:default\_inertial z\_value="${height3/2 - axle\_offset}" i\_value="1.0" mass="1"/>  
 </link>  
  
 <transmission name="transmission1">  
 <type>transmission\_interface/SimpleTransmission</type>  
 <joint name="joint\_base\_mid">  
 <hardwareInterface>hardware\_interface/EffortJointInterface</hardwareInterface>  
 </joint>  
 <actuator name="motor1">  
 <hardwareInterface>hardware\_interface/EffortJointInterface</hardwareInterface>  
 <mechanicalReduction>1</mechanicalReduction>  
 </actuator>  
 </transmission>  
  
 <transmission name="transmission2">  
 <type>transmission\_interface/SimpleTransmission</type>  
 <joint name="joint\_mid\_top">  
 <hardwareInterface>hardware\_interface/EffortJointInterface</hardwareInterface>  
 </joint>  
 <actuator name="motor2">  
 <hardwareInterface>hardware\_interface/EffortJointInterface</hardwareInterface>  
 <mechanicalReduction>1</mechanicalReduction>  
 </actuator>  
 </transmission>  
  
</robot>

* rrbot\_gazebo.launch:

<?xml version="1.0" encoding="UTF-8"?>  
<launch>  
 <!-- We resume the logic in gazebo\_ros package empty\_world.launch, -->  
 <!-- changing only the name of the world to be launched -->  
 <include file="$(find gazebo\_ros)/launch/empty\_world.launch">  
 <arg name="world\_name" value="$(find manipulator)/worlds/rrbot.world"/>  
   
 <arg name="paused" default="false"/>  
 <arg name="use\_sim\_time" default="true"/>  
 <arg name="gui" default="true"/>  
 <arg name="headless" default="false"/>  
 <arg name="debug" default="false"/>  
  
 </include>  
  
 <!-- Load the URDF into the ROS Parameter Server -->  
 <param name="robot\_description"  
 command="$(find xacro)/xacro '$(find manipulator)/urdf/rrbot.xacro'" />  
  
 <!-- Spawn rrbot into Gazebo -->  
 <node name="spawn\_urdf" pkg="gazebo\_ros" type="spawn\_model" respawn="false" output="screen"  
 args="-param robot\_description -urdf -model rrbot" />  
  
</launch>

1. **RESULT**: Successfully created a 3 joint manipulator, and simulated in Gazebo

# 5. Image Processing using OpenCV + ROS

1. **AIM**: To find the distance of image from origin in ROS.
2. **METHOD**:

* Save your ros distribution in a variable export ROS\_DIST=<noetic|melodic|kinetic>
* Install USB Camera support in ROS sudo apt-get install ros-$ROS\_DIST-usb-cam
* Install OpenCV sudo apt-get install opencv-python
* Verify Installations are in $PATH
* Start roscore: roscore
* Launch USB Camera: rosrun usb\_cam usb\_cam\_node
* Run the opencv script: rosrun exp5\_postlab image\_detect.py

1. **CODE**:

* image\_detect.py:

#!/usr/bin/env python3  
  
import rospy  
import cv2  
import numpy as np  
from std\_msgs.msg import String  
from sensor\_msgs.msg import Image, CameraInfo  
from cv\_bridge import CvBridge, CvBridgeError  
  
  
bridge = CvBridge()  
frame = None  
  
  
def blob\_detect(image, hsv\_min, hsv\_max, blur=0, blob\_params=None, imshow=False):  
 if blur > 0:  
 image = cv2.blur(image, (blur, blur))  
  
 if imshow:  
 cv2.imshow("Blur", image)  
 cv2.waitKey(0)  
  
 # BGR to HSV  
 hsv = cv2.cvtColor(image, cv2.COLOR\_BGR2HSV)  
  
 # - Apply HSV threshold  
 mask = cv2.inRange(hsv, hsv\_min, hsv\_max)  
  
 # - Show HSV Mask  
 if imshow:  
 cv2.imshow("HSV Mask", mask)  
  
 # - dilate makes the in range areas larger  
 mask = cv2.dilate(mask, None, iterations=2)  
  
 if imshow:  
 cv2.imshow("Dilate Mask", mask)  
 cv2.waitKey(0)  
  
 mask = cv2.erode(mask, None, iterations=2)  
  
 # - Show dilate/erode mask  
 if imshow:  
 cv2.imshow("Erode Mask", mask)  
 cv2.waitKey(0)  
  
 if blob\_params is None:  
 # Set up the SimpleBlobdetector with default parameters.  
 params = cv2.SimpleBlobDetector\_Params()  
  
 # Change thresholds  
 params.minThreshold = 0;  
 params.maxThreshold = 50;  
  
 # Filter by Area.  
 params.filterByArea = True  
 params.minArea = 5000  
 params.maxArea = 500000  
  
 # Filter by Circularity  
 params.filterByCircularity = True  
 params.minCircularity = 0.1  
  
 # Filter by Convexity  
 params.filterByConvexity = True  
 params.minConvexity = 0.5  
  
 # Filter by Inertia  
 params.filterByInertia = True  
 params.minInertiaRatio = 0.5  
  
 else:  
 params = blob\_params  
  
 # - Apply blob detection  
 detector = cv2.SimpleBlobDetector\_create(params)  
  
 # Reverse the mask: blobs are black on white  
 reversemask = 255 - mask  
  
 if imshow:  
 cv2.imshow("Reverse Mask", reversemask)  
 cv2.waitKey(0)  
  
 keypoints = detector.detect(reversemask)  
  
 return keypoints, reversemask  
  
  
def draw\_keypoints(image, keypoints, line\_color=(255, 0, 255), imshow=True):  
 im\_with\_keypoints = cv2.drawKeypoints(image, keypoints, np.array([]), line\_color,  
 cv2.DRAW\_MATCHES\_FLAGS\_DRAW\_RICH\_KEYPOINTS)  
  
 if imshow:  
 cv2.imshow("Keypoints", im\_with\_keypoints)  
  
 return (im\_with\_keypoints)  
  
  
def draw\_window(image, window\_adim, color=(255, 0, 0), line=5, imshow=False):  
 rows = image.shape[0]  
 cols = image.shape[1]  
  
 x\_min\_px = int(cols \* window\_adim[0])  
 y\_min\_px = int(rows \* window\_adim[1])  
 x\_max\_px = int(cols \* window\_adim[2])  
 y\_max\_px = int(rows \* window\_adim[3])  
  
 # -- Draw a rectangle from top left to bottom right corner  
 image = cv2.rectangle(image, (x\_min\_px, y\_min\_px), (x\_max\_px, y\_max\_px), color, line)  
  
 if imshow:  
 cv2.imshow("Keypoints", image)  
  
 return (image)  
  
  
def apply\_search\_window(image, window\_adim=[0.0, 0.0, 1.0, 1.0]):  
 rows = image.shape[0]  
 cols = image.shape[1]  
 x\_min\_px = int(cols \* window\_adim[0])  
 y\_min\_px = int(rows \* window\_adim[1])  
 x\_max\_px = int(cols \* window\_adim[2])  
 y\_max\_px = int(rows \* window\_adim[3])  
  
 # --- Initialize the mask as a black image  
 mask = np.zeros(image.shape, np.uint8)  
  
 # --- Copy the pixels from the original image corresponding to the window  
 mask[y\_min\_px:y\_max\_px, x\_min\_px:x\_max\_px] = image[y\_min\_px:y\_max\_px, x\_min\_px:x\_max\_px]  
  
 # --- return the mask  
 return (mask)  
  
  
def get\_blob\_relative\_position(image, keyPoint):  
 rows = float(image.shape[0])  
 cols = float(image.shape[1])  
 # print(rows, cols)  
 center\_x = 0.5 \* cols  
 center\_y = 0.5 \* rows  
 # print(center\_x)  
 x = (keyPoint.pt[0] - center\_x) / (center\_x)  
 y = (keyPoint.pt[1] - center\_y) / (center\_y)  
 return (x, y)  
  
  
  
def image\_callback(ros\_image):  
 global frame  
 try:  
 frame = bridge.imgmsg\_to\_cv2(ros\_image, "bgr8")  
 cv2.imshow("image", frame)  
 except CvBridgeError as e:  
 print(e)  
  
 lower\_color = (36, 25, 25)  
 higher\_color = (70, 255, 255)  
  
 string1 = "first frame"  
 string2 = ""  
 string3 = ""  
 window = [0, 0, 1, 1]  
  
 keypoints, \_ = blob\_detect(frame, lower\_color, higher\_color, blur=3, blob\_params=None, imshow=False)  
 final\_image = draw\_keypoints(frame, keypoints, imshow=True)  
 cv2.putText(final\_image, string1, (25, 25), cv2.FONT\_HERSHEY\_COMPLEX, 1, (255, 0, 0), 1)  
 cv2.putText(final\_image, string2, (25, 50), cv2.FONT\_HERSHEY\_COMPLEX, 1, (255, 0, 0), 1)  
 cv2.putText(final\_image, string3, (25, 75), cv2.FONT\_HERSHEY\_COMPLEX, 1, (255, 0, 0), 1)  
 cv2.imshow("final image", final\_image)  
 for i, keyPoint in enumerate(keypoints):  
 x = keyPoint.pt[0]  
 y = keyPoint.pt[1]  
 s = keyPoint.size  
 print("kp %d: s = %3d x = %3d y= %3d" % (i, s, x, y))  
  
 x, y = get\_blob\_relative\_position(frame, keyPoint)  
 print(" x = %3d y= %3d" % (x, y))  
 string1 = 'x=' + str(x)  
 string2 = 'y=' + str(y)  
 string3 = 's=' + str(s)  
  
 key = cv2.waitKey(1) & 0xFF  
 if key == ord('q'):  
 return  
  
  
  
  
  
  
def listener():  
 rospy.Subscriber("/usb\_cam/image\_raw", Image, image\_callback)  
  
 rospy.spin()  
 cv2.destroyAllWindows()  
  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 try:  
 rospy.init\_node('ar\_tracking\_node', anonymous=True, disable\_signals=True)  
 rate = rospy.Rate(100)  
 listener()  
  
 except rospy.ROSInterruptException:  
 pass

1. **RESULT**: Successfully implemented blue cap detection via opencv + ros