uOttawa

Faculty of Engineering
MCG 5353 – Robotics
Spring/Summer 2023
Dr. Amirhossein Monjazeb, P.Eng

Design and Development of a Robotic Sorting System for A Production Line

#	Group member's full last and first name	UOIT student ID number	Signature
1	Mariam Hussien	300389400	Mariam Hussien Ali
2	Ibrahim Elshenhapy	300389386	Ibrahim Elshenhapy
3	Ahmed Asem	300389894	AHMED Attya
4	Ghada Salah	300389364	Ghada Saleh
5	Shady Osama	300389887	Shady

Contents

Introduction

ROS (Robot Operating System) is an open-source framework for building robotic systems. It enables communication between software components, offers modularity, and supports inter-process communication and sensor integration. ROS organizes functionality into packages containing nodes that communicate through topics or services. Gazebo is a 3D physics-based simulator integrated with ROS, ideal for testing and developing robots in a virtual environment before real-world deployment.

In this project, our group will create a Robotic Sorting System for a big production line. The system will use a robotic arm in a computer simulator called Gazebo. We'll be dealing with different types of bricks with various shapes and colors that come on a conveyor belt. Our task is to make the robot identify at least three types of bricks, pick them up from the conveyor, and place them in specific areas for packing.

Objective

- Create a Sorting Station: Make a robot-based sorting station in the Gazebo simulator for the production line.
- Detect Different Bricks: Teach the robot to recognize at least three types of bricks as they move on the conveyor.
- Pick Up Bricks Carefully: Program the robot to pick up the recognized bricks carefully from the conveyor.
- Place Bricks in Designated Areas: Make the robot put the picked bricks in their assigned places for packing.
- Test and Improve: Test the robot's sorting abilities and make any needed improvements to ensure it works well.

Requirements

1. Modeling the Robot and Embedding Motors and Sensors: We'll design or find a suitable file (URDF or Xacro) for the robot with 6 or 7 degrees of freedom. Making sure the simulator's dimensions match the real robot is crucial. We'll also add motors and sensors to the robot's joints and links for accurate simulation in Gazebo.
2. Workspace Design: Creating a well-designed workspace in Gazebo with a conveyor and essential components is vital for the sorting station's success.
3. Robot Control with ROS: We'll use ROS controllers to control the robot's joints, ensuring precise and efficient movement.
4. Object Position Detection and Command Generation: To detect brick positions in Gazebo.

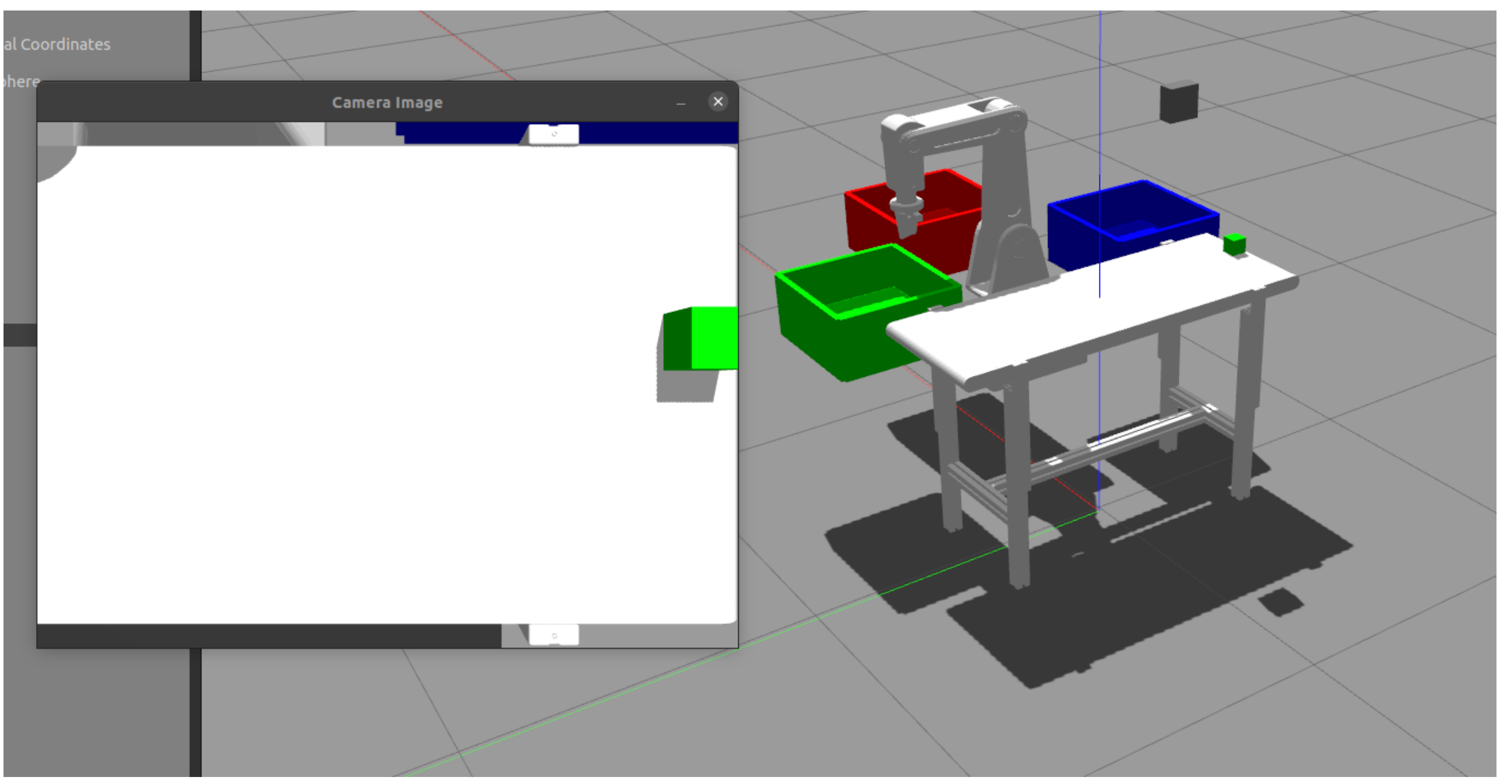

Figure 1 Our Workspace as starting.

1- Modeling the Robot and Embedding Motors and Sensors:

a. Create or find a URDF or xacro file for your desired robot model, ensuring that the dimensions in the simulator match the real robot.

We found a robot online that was designed and transformed into a URDF file and started working from there. The robot is shown in Figure (2).

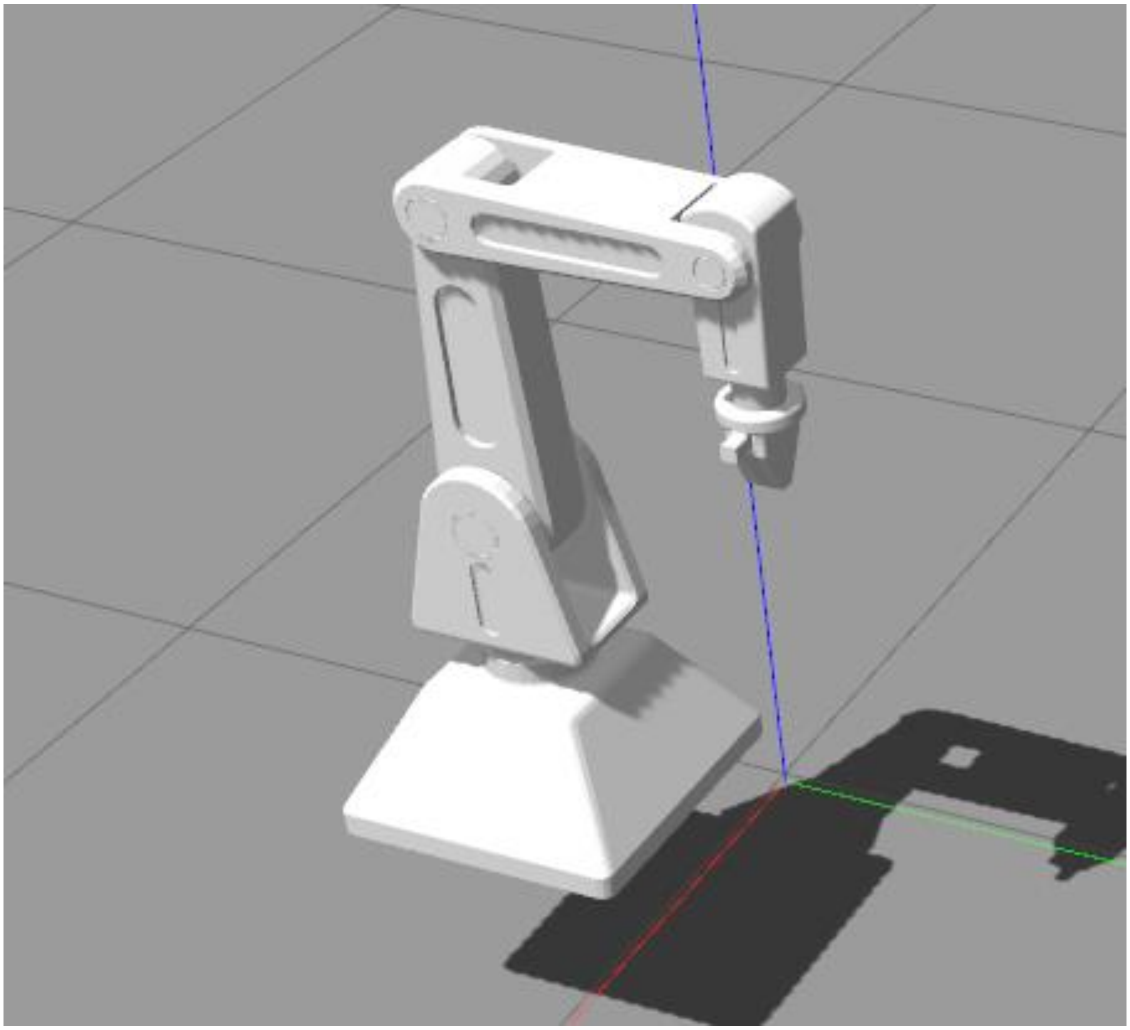

Figure 2 Robot arm

b. Attach motors to the joints and sensors to the links for the simulation of robot in Gazebo.

First, we edited the package.xml and cmake files of the robot URDF to add the dependencies for the robot to work correctly.

```
<transmission name="link_1_trans">
      <type>transmission_interface/SimpleTransmission</type>
      <joint name="joint_1">
              <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
      </joint>
      <actuator name="link_1_motor">
              <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
              <mechanicalReduction>1</mechanicalReduction>
      </actuator>
</transmission>
```

Figure 3 Attach Motor to joint 1 (robot_arm_urdf.urdf)

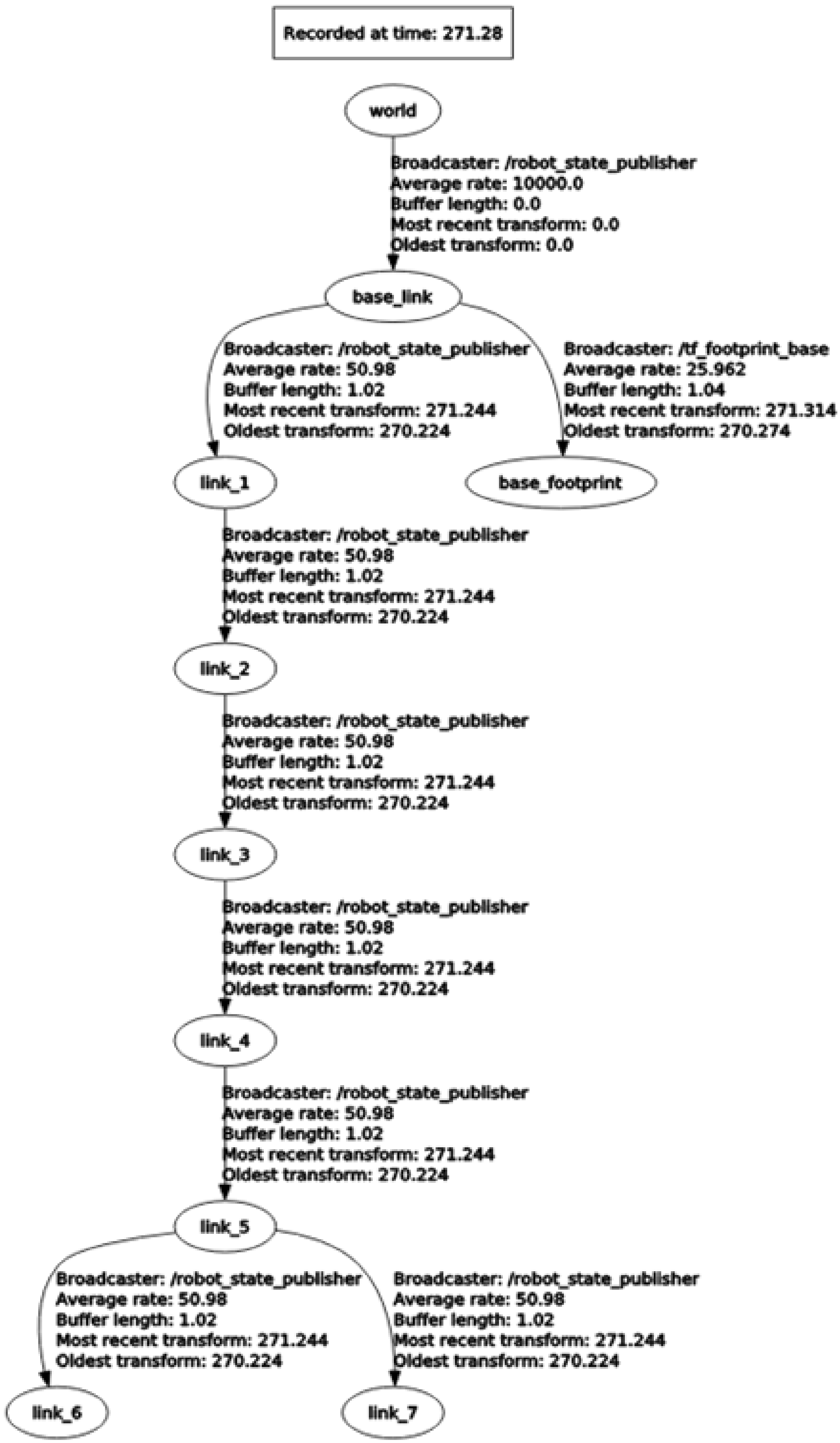

Figure 4 Robot links in a time instant.

2- Workspace Design:

a. Design the workspace in Gazebo, including a conveyor and any other necessary components for the sorting station.

The simulation is on Gazebo where I can control the virtual environment of the robot. We start by creating a new empty world. Every empty world has orthogonal coordinates (x, y, z) where we can refer to every object in that environment to its origin point. In the empty world some attributes such as a light source and gravity are implemented to mimic the physical world. We used the gazebo ROS library as a reference for the empty world used.

```
<arg name="paused" default="false"/>
<arg name="gazebo_gui" default="true"/>
<arg name="urdf_path" default="$(find robot_arm_urdf)/urdf/robot_arm_urdf.urdf"/>

<!-- startup simulated world -->
<include file="$(find gazebo_ros)/launch/empty_world.launch">
  <arg name="world_name" default="worlds/empty.world"/>
  <arg name="paused" value="$(arg paused)"/>
  <arg name="gui" value="$(arg gazebo_gui)"/>
</include>
```

Figure 5 Launching Gazebo empty world (full_robot_arm_sim.launch)

Then, in this world we launch each object using different ways such as urdf files. The objected launched is in reference to the world origin point. We can control the position and the orientation using 6 factors: x , y , z , raw , pitch , yaw. We launched a gazebo conveyor plugin that helped us control the conveyor and spawn cubes on it. The file uses a reference urdf files that represent the conveyor, red cube , green cube , and blue cube. The demo.py file in the demo world package.

```python
def __init__(self) -> None:
    self.rospack = rospkg.RosPack()
    self.path = self.rospack.get_path('demo_world')+"/urdf/"
    self.cubes = []
    self.cubes.append(self.path+"red_cube.urdf")
    self.cubes.append(self.path+"green_cube.urdf")
    self.cubes.append(self.path+"blue_cube.urdf")
    self.col = 0

    self.sm = rospy.ServiceProxy("/gazebo/spawn_urdf_model", SpawnModel)
    self.dm = rospy.ServiceProxy("/gazebo/delete_model", DeleteModel)
    self.ms = rospy.ServiceProxy("/gazebo/get_model_state", GetModelState)
```

Figure 6 Spawning function initialization (demo.py)

Then we added the boxes of the environment where each box is used for a certain cube color.

```
<node name="blue_box" pkg="gazebo_ros" type="spawn_model" args="-file $(find demo_world)/urdf/bluebox.urdf -urdf -model blue_box -x 0.5 -y -0.5 -z 0.6 " respawn="false" output="screen" />

<node name="green_box" pkg="gazebo_ros" type="spawn_model" args="-file $(find demo_world)/urdf/greenbox.urdf -urdf -model green_box -x 0.5 -y 0.5 -z 0.6" respawn="false" output="screen" />

<node name="red_box" pkg="gazebo_ros" type="spawn_model" args="-file $(find demo_world)/urdf/redbox.urdf -urdf -model red_box -x 1 -y 0.0 -z 0.6" respawn="false" output="screen" />
```

Figure 7 Launching the sorting boxes. (full_robot_arm_sim.launch)

Launching The conveyor with random generated boxes with (blue, red and green) color in Gazebo world to be as shown in the next figure. We already have the 3 boxes with the conveyor and blocks as shown in figure (1) and will be later shown as the system working.

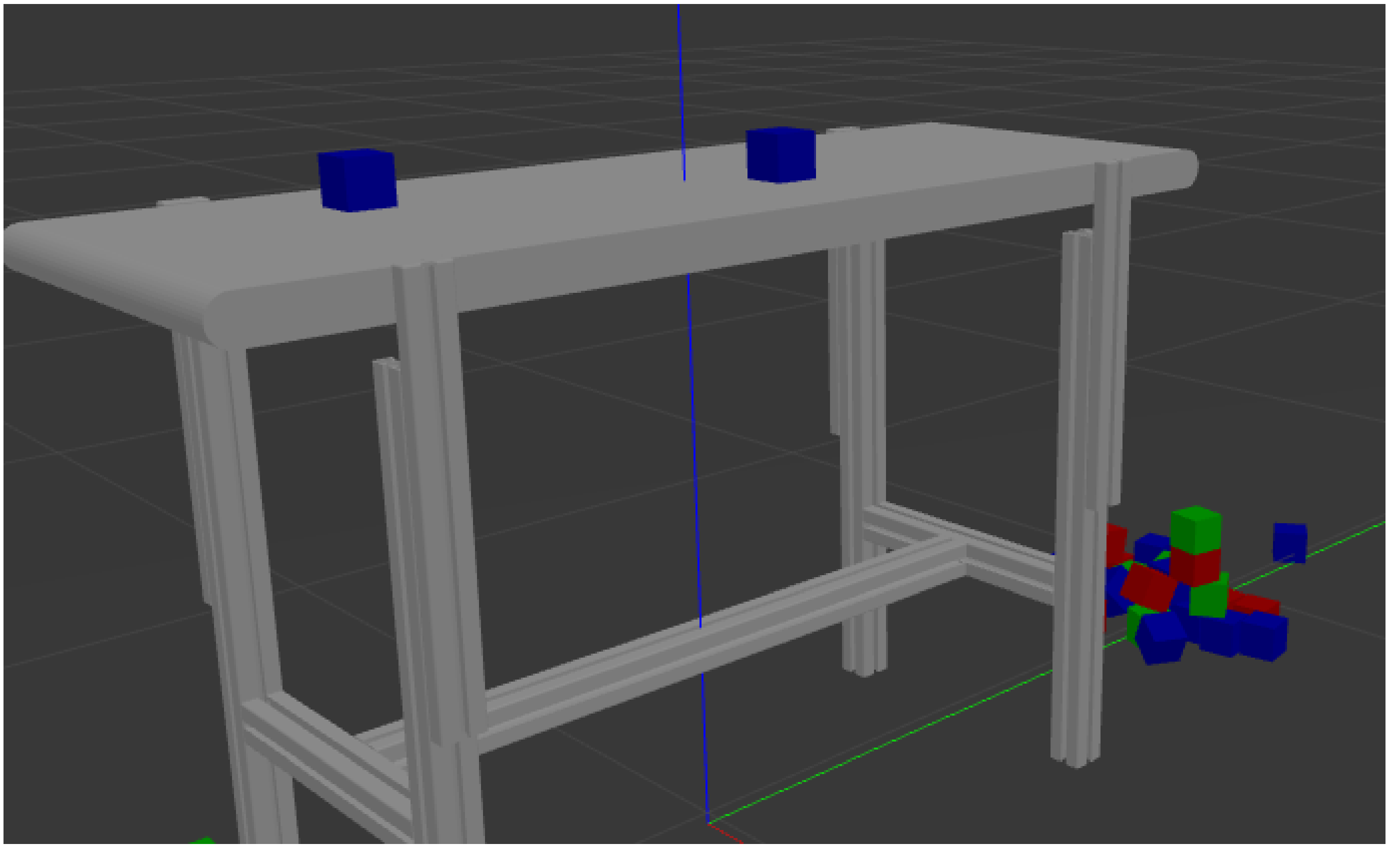

Figure 8 Moving conveyor with randomly spawned cubes.

b. Justify your design choices and the selection of components in your final report.

In our final report, we justified the design choices and component selections for the conveyor system based on several factors that are essential to the success of our project. Here are the key justifications:

- Availability of Gazebo Conveyor Plugin: The selection of the conveyor system with the Gazebo Conveyor Plugin was primarily driven by its availability and compatibility with our simulation environment. Gazebo is a widely used robotics simulation platform, and the existence of a conveyor plugin allowed us to easily integrate the conveyor system into our simulation.
- Convenient Cube Sizes: The choice of cube sizes was based on their convenience to our robot arm's end effector size. By selecting cube sizes that match or are close to the dimensions of the robot's gripper, we can ensure that the robot can effectively pick up and place the cubes without any collision or gripping issues.
- Boxes are clear Visibility for Item Placement: The decision to use boxes for the conveyor system was made because boxes provide a clear and well-defined space for placing items. The open-top design of the boxes allows us to easily position items inside them, making it convenient for the robot to handle and manipulate the objects.
- Suitable Robot Workspace: The selection of the robot arm with its specific workspace was crucial to ensure that the robot could reach all the desired places on the conveyor system. By analyzing the robot's workspace and comparing it with the layout of the conveyor system, we ensured that the robot can efficiently handle the objects on the conveyor.

3- Robot Control

a. Implement ROS controllers to control the robot's joints.

We added the required controllers as you can find in the next figure.

```
# Creates the /joint_states topic necessary in ROS
joint_state_controller:
  type: joint_state_controller/JointStateController
  publish_rate: 50
controller_list:
  - name: robot_arm_controller
    action_ns: follow_joint_trajectory
    default: True
    type: FollowJointTrajectory
    joints:
      - joint_1
      - joint_2
      - joint_3
      - joint_4
      - joint_5
  - name: hand_ee_controller
    action_ns: follow_joint_trajectory
    default: True
    type: FollowJointTrajectory
    joints:
      - joint_6
      - joint_7
hand_ee_controller:
  type: effort_controllers/JointTrajectoryController
  joints:
    - joint_6
    - joint_7
```

Figure 9 Ros Controllers for robot joints (ros_controllers.yaml)

b. Save the PID parameters in a YAML file for future runs.

Our Pid is done for all joints you can find part from the pid gains for the robot arm in the next figure.

```
/gazebo_ros_control:
pid_gains:
  joint_2:
    p: 100
    d: 1
    i: 1
    i_clamp: 1

/gazebo_ros_control:
pid_gains:
  joint_3:
    p: 100
    d: 1
    i: 1
    i_clamp: 1
```

Figure 10 PID gains for the joints 2,3 (ros_controllers.yaml)

c. Create launch files that run all nodes and parameters, enabling a complete project launch.

We collect all in one launch file for the arm, the conveyor, cubes, collecting boxes and the camera.

```
C: > Users > Shenhapy > Downloads > Compressed > src > src > import_your_custom_urdf_package_to_ROS-main-2e713d1acf99981a315667f32bbb82a
<launch>
    <!-- Launch Your robot arms launch file which loads the robot in Gazebo and spawns the controllers -->
    <include file = "$(find robot_arm_urdf)/launch/arm_urdf.launch" />

    <!-- Launch Moveit Move Group Node -->
    <include file = "$(find movit_robot_arm_sim)/launch/move_group.launch" />

    <!-- Run Rviz and load the default configuration to see the state of the move_group node -->
    <arg name="use_rviz" default="true" />
    <include file="$(find movit_robot_arm_sim)/launch/moveit_rviz.launch" if="$(arg use_rviz)">
        <arg name="rviz_config" value="$(find movit_robot_arm_sim)/launch/moveit.rviz"/>
    </include>

        <group ns="conveyor">
        <node name="spawn_model_belt" pkg="gazebo_ros" type="spawn_model" args="-file $(find demo_world)/ur
        </group>

        <node name="blue_box" pkg="gazebo_ros" type="spawn_model" args="-file $(find demo_world)/urdf/blueb

<node name="green_box" pkg="gazebo_ros" type="spawn_model" args="-file $(find demo_world)/urdf/greenbox.urd

<node name="red_box" pkg="gazebo_ros" type="spawn_model" args="-file $(find demo_world)/urdf/redbox.urdf -u

    <node name="cube_spawner" pkg="demo_world" type="demo.py" output="screen"/>


  <!-- Spawn the camera model -->
  <node name="spawn_camera" pkg="gazebo_ros" type="spawn_model" args="-sdf -file $(find movit_robot_arm_sim

</launch>
```

Figure 11 Launch file for whole project (full_robot_arm_sim.launch)

4- Object Position Detection and Command Generation:

a. Develop a node to obtain the positions of bricks from the Gazebo simulator.

We publish the position of each brick as the cube on the conveyor.

```python
    def spawnCubes(self, num_cubes):
        for _ in range(num_cubes):
            if self.checkModel() == False:
                self.spawnModel()
                x,y,z = self.getPosition()
                pub = rospy.Publisher('chatter', String, queue_size=10)
                rate = rospy.Rate(10) # 10hz
                while z >= 0.7:
                    hello_x = "x position = %s \n" % x
                    hello_y = "y position = %s \n" % y
                    hello_z = "z position = %s \n" % z
                    # rospy.loginfo(hello_x)
                    # rospy.loginfo(hello_y)
                    # rospy.loginfo(hello_z)
                    x,y,z = self.getPosition()
                    pub.publish(hello_x)
                    pub.publish(hello_y)
                    pub.publish(hello_z)
                    rate.sleep()
```

Figure 12 Publish cubes positions (demo.py)

We now will subscribe to the cube position so that we later give the robot arm it's location to be able to pick.

```python
def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
```

```python
    rospy.init_node('__init__', anonymous=True)
    rospy.Subscriber("chatter", String, callback)
    desired = float(callback.data)
```

Figure 13 Subscribe the cube position (move_to_point.py)

b. Generate commands for each controller to enable the robot to pick and place bricks in the simulation.

First, we define arm group and gripper group

```python
    group_name = "arm_group"
    group = moveit_commander.MoveGroupCommander(group_name)
    gripper_group = moveit_commander.MoveGroupCommander('hand')
```

Figure 14 Define arm and gripper group (move_to_point.py)

Now we go to the initial position.

```python
def go_to_joint_state(self):

    group = self.group

    joint_goal = group.get_current_joint_values()
    joint_goal[0] = pi/2
    joint_goal[1] = 0
    joint_goal[2] = 0
    joint_goal[3] = 0
    joint_goal[4] = 0

    group.go(joint_goal, wait=True)

    group.stop()

    current_joints = self.group.get_current_joint_values()
    return all_close(joint_goal, current_joints, 0.01)
```

Figure 15 Intial position for the robot (move_to_point.py)

Using this function, we go to the goal position which obtained from the subscribe node.

```python
def go_to_pose_goal(self,xyzrpy):

  group = self.group

  pose_goal = geometry_msgs.msg.Pose()
  pose_goal.orientation.x = xyzrpy[3]
  pose_goal.orientation.y = xyzrpy[4]
  pose_goal.orientation.z = xyzrpy[5]
  pose_goal.orientation.w = 0.0

  pose_goal.position.x = xyzrpy[0]
  pose_goal.position.y = xyzrpy[1]
  pose_goal.position.z = xyzrpy[2]
  group.set_joint_value_target(pose_goal,True)

  plan = group.go(wait=True)
  group.stop()
  group.clear_pose_targets()

  current_pose = self.group.get_current_pose().pose
  return all_close(pose_goal, current_pose, 0.01)
```

In the init function of the class we use this function to get to the goal

```python
self.go_to_joint_state()

xyz = [0.0, desired, 0.0]
rpy = self.rad([0, 0, 0])
xyzrpy = xyz+rpy
self.go_to_pose_goal(xyzrpy)

# Open the gripper and deactivate it
gripper_group.go([0.03, -0.03], wait=True)  # Deactivate gripper
gripper_group.stop()

# Decrease z to reach object
xyz = [0.0, 0.0, -0.2]
rpy = self.rad([0, 0, 0])
xyzrpy = xyz+rpy
self.go_to_pose_goal(xyzrpy)

# Activate the gripper and close it
gripper_group.go([0.018,-0.018], wait=True)  # Activate gripper
gripper_group.stop()

# Go to box position
xyz = [0.0, 0.0, -0.2]
rpy = self.rad([0, 0, 0])
xyzrpy = xyz+rpy
self.go_to_pose_goal(xyzrpy)

# Open the gripper and deactivate it to throw in the box
gripper_group.go([0.03, -0.03], wait=True)  # Deactivate gripper
gripper_group.stop()
```

As the robot arm receives the cube position it starts to move to pick it then place it in the box where the camera used for cube color detection to have the decision of with box to place the cube in the right box.

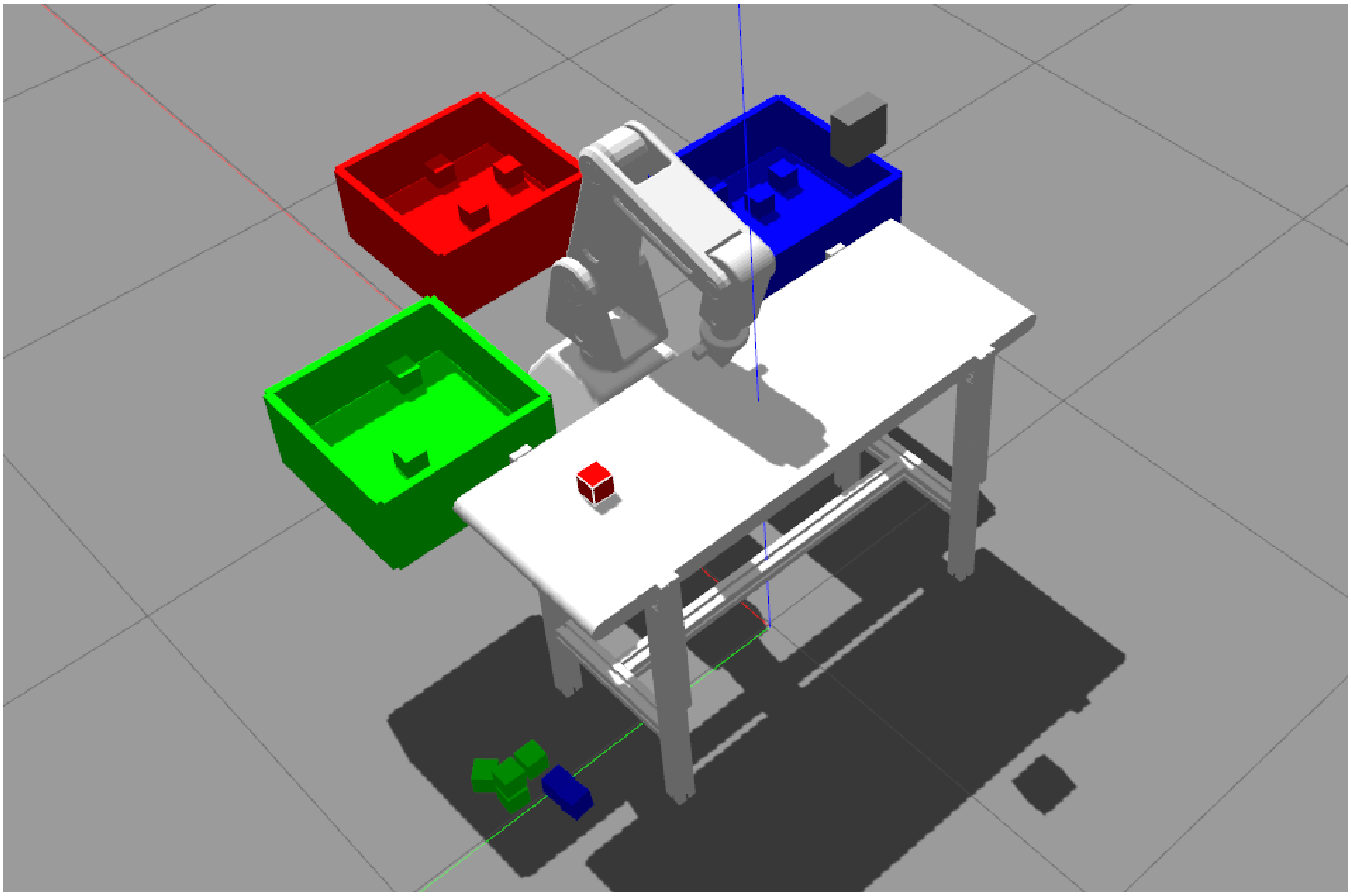

Figure 16 Robot after placing some cubes and missing some.

5- Extra Credit

a. Install a camera on the ceiling to capture top-down photos in the simulation. Implement image processing techniques to detect object positions and types instead of relying solely on available link positions published with Gazebo.

We installed the camera into the system. The camera represents a .sdf file that has a box as a camera sensor with the RGB format. The sdf file has a plugin that represents the camera parameters and the information about the topic the images is published to it.

```
<plugin name="camera_controller" filename="libgazebo_ros_camera.so">
  <!-- The camera plugin configuration goes here -->
  <alwaysOn>true</alwaysOn>
  <updateRate>30.0</updateRate>
  <cameraName>camera</cameraName>
  <imageTopicName>/camera_model/camera/image_topic</imageTopicName>
  <cameraInfoTopicName>camera_info_topic</cameraInfoTopicName>
  <frameName>camera_frame</frameName>
  <hackBaseline>0.07</hackBaseline>
  <distortionK1>0.0</distortionK1>
  <distortionK2>0.0</distortionK2>
  <distortionK3>0.0</distortionK3>
  <distortionT1>0.0</distortionT1>
  <distortionT2>0.0</distortionT2>
</plugin>
```

Figure 17 Launching the sorting boxes (mycam.sdf)

Then we added the image subscriber node that subscribes on the /camera_model/camera/image_topic in order to access the recent camera RGB image captured by the camera sensor. This enabled us to have a live capture of the top view image through the camera.

```
def image_callback(data):

    try:
        # Convert ROS Image message to OpenCV format
        cv_image = CvBridge().imgmsg_to_cv2(data, "bgr8")
        cv2.imshow("Camera Image", cv_image)

        cv2.waitKey(1)  # Refresh display (1 millisecond delay)
    except Exception as e:
        rospy.logerr(e)

def image_subscriber():
    rospy.init_node('image_subscriber', anonymous=True)
    rospy.Subscriber('/camera_model/camera/image_topic', Image, image_callback)
    rospy.spin()
```

Figure 18 Image Callback (cam_subscriber.py)

To conclude we used those items as an environment for our station each one for a reason:

- Empty world: to have a space with an orthogonal representation.
- Conveyor: to have objects to move on, to keep the process running and automated.
- Cubes different colors: the targeted objects to be sorted.
- Robotic arm to be able to reach out for the cube in a proper orientation and move pick the object and place it in the right spot.
- Boxes: to sort the cubes in.
- Camera: to sort the cubes by color using open cv and trigger a callback once the cube is in the right spot.

Then we added a mask for all the ranges of boxes colors we might need. We converted the RGB into HSV as it represents a better color map when it comes to classification. After that, we added masks for (green and lemon green), (blue and light blue), and red. Then we published the results on a topic to use it to give the action to the robot arm to start the pick and place for that specific scenario.

```
# Red
lower_red1 = np.array([0, 100, 100])
upper_red1 = np.array([10, 255, 255])

lower_red2 = np.array([160, 100, 100])
upper_red2 = np.array([179, 255, 255])

# Green
lower_green = np.array([40, 40, 40])
upper_green = np.array([80, 255, 255])

# Lemon Green
lower_lemon_green = np.array([25, 40, 40])
upper_lemon_green = np.array([45, 255, 255])

# Blue
lower_blue = np.array([100, 100, 100])
upper_blue = np.array([140, 255, 255])

# Light Blue
lower_light_blue = np.array([80, 40, 40])
upper_light_blue = np.array([100, 255, 255])

# Convert the image to the HSV color space
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
```

Figure 19 Masking the colors (Classifier.py)

As previously explained, the camera sdf model has an installed plugin that captures gazebo environment in RGB formal and publishes it as an image type message on the designed topic: /camera_model/camera/image_topic. Then, we subscribed to this node to both view the image as a live view and to classify the image of the cube color.

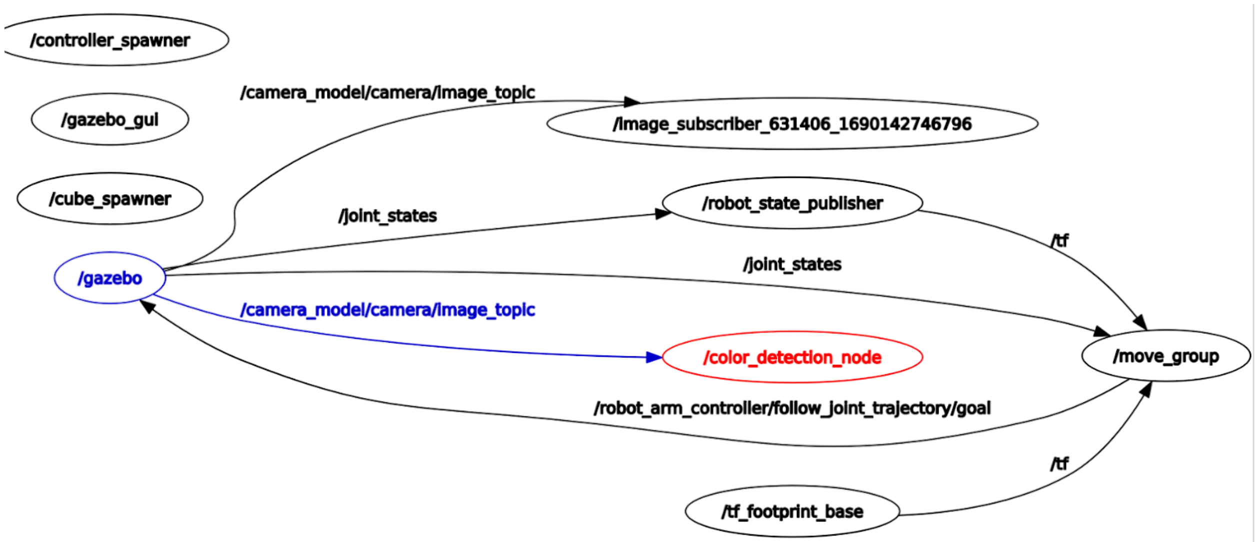

Figure 20 Color detection node

As seen here, the camera can see the green cube at the starting of the conveyor from the top view which is shown in the camera image window. We can run this file by running the cam_subscriber.py in the Moveit_robot_arm_sim package.

Future Work

- Hardware Selection: Choose an appropriate robotic arm with the required degrees of freedom and payload capacity for real-world application. Consider factors such as cost, availability, and compatibility with the system.
- Sensor Integration: Integrate real-world sensors (e.g., cameras, depth sensors) on the physical robot to detect and identify the different types of bricks on the conveyor.
- Real-Time Control: Develop real-time control algorithms and interfaces to ensure the robot can respond quickly and accurately to the changing environment of the production line.

Conclusion

In this group project, we designed and developed a robotic sorting system using the Gazebo simulator. The system can identify different types of bricks on the conveyor and place them in designated areas for packing. For future work, we could explore transitioning to real hardware integration, including selecting an appropriate robotic arm, integrating sensors. The project provided valuable practical experience in robotics, control systems, and simulation, fostering collaboration and problem-solving skills. It highlighted the potential of robotics in modern manufacturing and contributed to our understanding of automation applications.

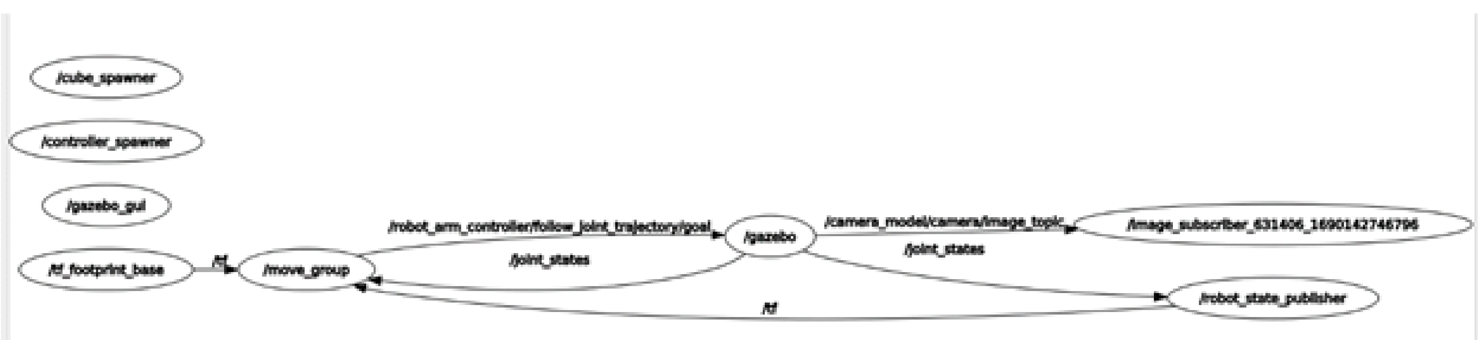

Figure 21 Moveit topic

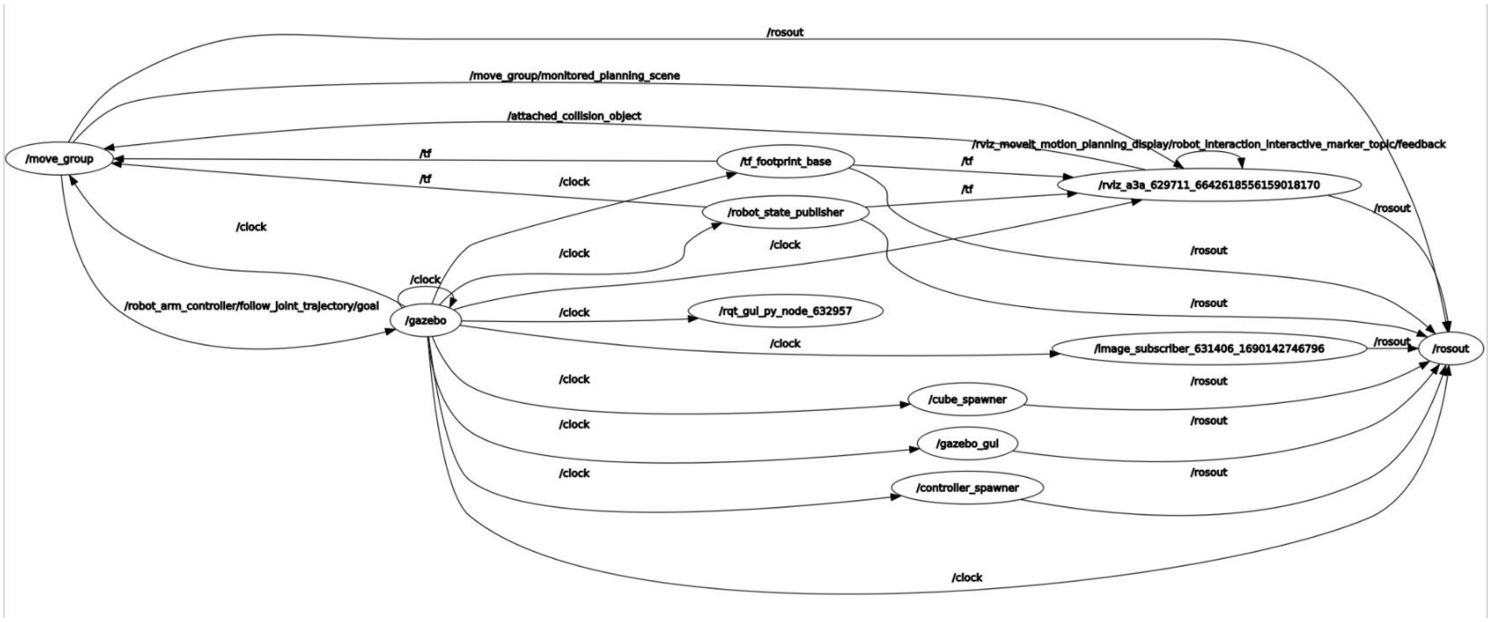

Figure 22 rqt graph

References

1. ROS Wiki: Documentation (https://wiki.ros.org/Documentation)
2. Gazebo Software and Documents (https://classic.gazebosim.org/tutorials)