Autonomous Pick-and-Place Sorting using a UR5 Robotic Manipulator

Abstract

This report demonstrates a proof-of-concept simulation setup for sorting of objects based on colour into bins using a UR5 Robotic Manipulator. The report covers an overview of robotic manipulators and their applications, and elaborates upon applications involving sorting and categorisation of objects with respect to the current implementation. The various criteria for analysis and the ways the results of this report may be extrapolated to real-life applications are also expanded on.

Introduction

Sorting objects into categories is a basic operation in any robotics or manufacturing application. Categorisation of objects based on colour, size, and shape, and the various processes to achieve this have long since been a focus on many robotics applications. Robotic manipulators are capable of performing repetitive tasks at speeds and accuracies that far exceed those of human operators. They are now widely used in manufacturing processes such as spot welding and painting [1].

Fig. 1: Robotic Manipulators performing sorting operations on a conveyor belt.

Literature

The importance of sorting objects in day-to-day lives

The process of sorting objects into various categories is required in various factories in automated processes like sorting of nuts and bolts at a mechanical station, segregation of the different types of garbage at a recycling plant and even sorting packaged foods as vegetarian/non-vegetarian based on the label colour (green/red). More precise machines can be used to sort coins, segregate clothes and pick out defective items from a conveyer belt. Table 1 enumerates the parameters and corresponding categories for each application in detail:

Table 1: Applications of sorting objects and their various considerations.

Application	Parameters to be considered	Categories
Sorting of nuts and bolts	Length, shape of head, size, type of thread	Hex-headed bolt, square-headed bolt, cylindrical bolt, ring nuts, cap nuts

Garbage segregation	Material	Biodegradable and non-biodegradable
Sorting packaged foods	Colour of the label	Vegetarian and non-vegetarian
Sorting coins	Size, shape, colour	Currency-dependent. Example: ₹1, ₹2, ₹5 and ₹10 coins
Segregate clothes	Shape, texture, colour	Shirts, T-shirts, Sweaters, Socks, Jeans
Pick out defective items from a conveyor	Product-dependent, but has one parameter like electrical conductivity, for example, varying in defective items	Defective and not defective

Safety and Ethics

Safety has to be taken into consideration every time any automated process is deployed at a workplace. Appropriate training needs to be given to the workforce on safe operational practices. Risk assessments have become mandatory as part of the Robotic Industries Association's R15.06-2013 standard and they are one of the best practices to be conducted regularly. A thorough risk assessment can find hazards not only relating to the robot motion, but also flying debris and dangerous substances or chemicals. Light curtains, laser scanners and other presence-sensing devices are a commonly used and widely accepted method of machine guarding in manufacturing facilities. A fast-acting automated barrier door or roll-up curtain eliminates exposure to dangerous movement machines and hazards produced by the process, such as smoke, flash, splash, mist and flying debris [2].

Problem Statement

The problem statement tackled in this report is as follows:

> Perform pick-and-place operation-based sorting of coloured boxes on a table into colour-coded bins. A logical camera mounted above the table locates and categorises the boxes, and a UR5 Robotic Manipulator sorts the boxes by colour into their corresponding bins, located around the robot. Simulate this setup using ROS and any essential setups/libraries/functions as per requirement.

The intended objectives to be achieved in the course of the problem statement are:

1. UR5 Motion Planning using MoveIt! and RViz;
2. Integration of Logical Cameras [3] in Gazebo to detect and categorise objects; and
3. ROS setup to oversee the entire process.

Methodology

Overview

The implementation described in this report follows the following method of execution:

1. Create a simulation environment holding all object assets to visualise the implementation;
2. Integrate a camera-connected-esque interface in the simulation environment to process information received from it and thus locate the boxes on the table; and

3. Create a control system to act upon the data received from the camera interface and control the UR5 to perform the task.

The following ROS integrations and resources [4][5][6][7] were utilised to achieve this:

Gazebo

Gazebo is the primary simulation environment for ROS, featuring a robust physics engine, high-quality graphics, and programmatic and graphical interfaces to help simulate ROS implementations. By creating environments for specific use-cases (for example, city roads for a self-driving car or a factory line for an industrial robot), we can observe how the robot autonomously interacts with its environment.

RViz

RViz is a robot 3D visualisation environment that enables viewing and processing data received from cameras, lasers, encoders and other sensors mounted atop a robot, thus allowing us to view the world through the robot's eyes and better understand the way it observes and can interact with its environment.

MoveIt!

MoveIt! is a robotics manipulation platform that primarily enables motion planning algorithms and overall task planning for robotics applications. It integrates the capabilities of Gazebo and RViz on a single platform along with controllers for robotic manipulation and control to create a single integrated system that integrates with ROS to perform tasks.

URDF

Unified Robot Description Format (URDF) is an XML format for representing a robot model. URDFs are used to generate rigid body trees that represent the robot through links and joints be describing the properties of each link and their connections to each other. URDFs are essential to describing a robot in any environment and making sensible connections between robot controllers and real-life sensors on a robot.

Timeline

The implementation process is broken up into the following stages:

1. Building the Gazebo World and RViz Planning Scene
2. Setting up Logical Camera functionality with TF
3. Integration with MoveIt! and ROS Scripting

Building the Gazebo World

The Gazebo world in Fig. 1 contains the following objects:

- A UR5 Robotic Manipulator [9] mounted on a pedestal [8];
- A table [8] on which the boxes are placed;
- A logical camera [8] mounted above the table with the table in its field of view; and
- 3 bins [8], coloured red, blue and green respectively.

The table also contains 6 boxes [8] (2 red, 2 green, 2 blue) arranged randomly in front of the UR5.

The coloured boxes in Fig. 2 were spawned individually using a separate spawn script [11] after the Gazebo world creation to preserve their body frames, as including them in the Gazebo world from the beginning resulted in their body frames locking to the origin [12]. This resulted in the Gazebo link-attach function [11] not working – executing the function would make the box disappear from, and subsequently crash, the Gazebo simulation.

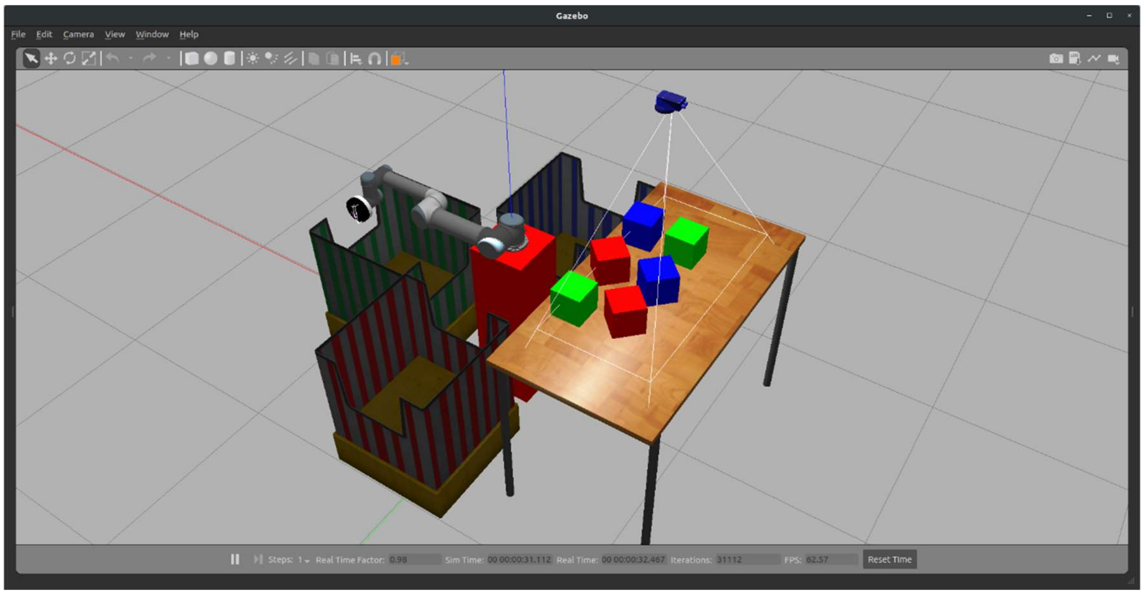

Gazebo
File Edit Camera View Window Help
Steps: 1 Real Time Factor: 0.98 Sim Time: 00 00:00:31.112 Real Time: 00 00:00:32.467 Iterations: 31112 FPS: 62.57
Reset Time

Fig. 2: The Gazebo World.

Building the RViz Planning Scene

The RViz Planning Scene in Fig. 2 contains the following objects:

- A UR5 Robotic Manipulator mounted on a pedestal;
- A placeholder object for the top of the table as seen in the Gazebo world; and
- 6 objects representing the boxes as seen in the Gazebo world.

The bins and logical camera in Fig. 3 were deemed non-essential to motion-planning requirements through observation due to the negligible effect they had on the results of the motion planning and subsequent results, and have thus not been included to streamline the setup.

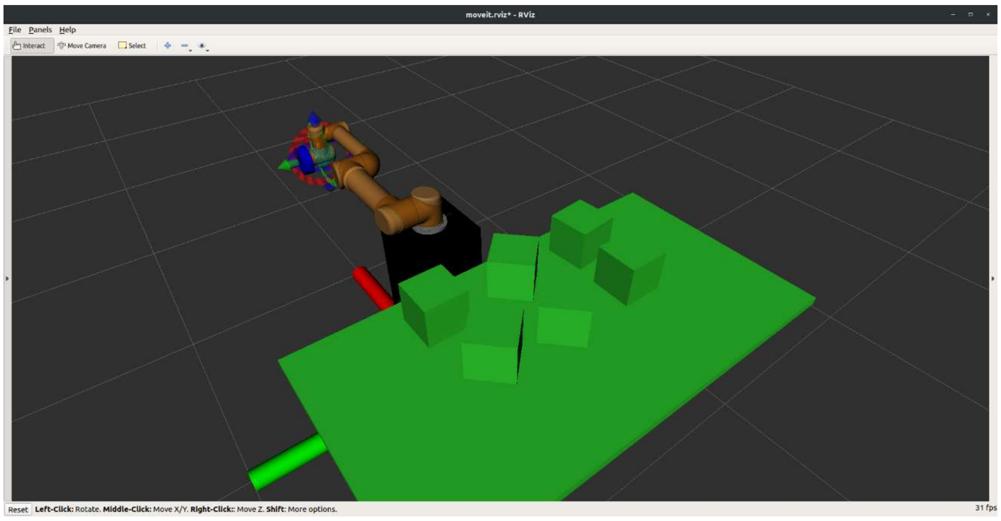

moveit.rviz* - RViz
File Panels Help
Interact Move Camera Select
Reset Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click:: Move Z. Shift: More options.
31 fps

Fig. 3: The RViz Planning Scene.

Setting up TF

TF ('TransForm') is the primary method for objects as detected through the logical camera to be visualised as frames in RViz. The logical camera visualises the frames of the boxes in the RViz Planning Scene [15] as shown in Fig, 4. This information can then be used to direct the UR5 to travel to the specified pose (the position of the specified box) for executing the pick operation.

The following process was implemented:

1. Use the Logical Camera to detect the boxes, extracting the list of the names of the models detected.
2. Determine which models contain package in their name, indicating that these models are the boxes to be picked.
3. Feed the colours, poses and orientations of the boxes one by one to the UR5, which will then perform the corresponding pick-place operation.

The logical camera also detects the UR5 and the table in Gazebo; these two entries are ignored whilst searching for all classifying the boxes being detected. This can extrapolate to any non-essential environmental objects which may pose a hindrance to data processing.

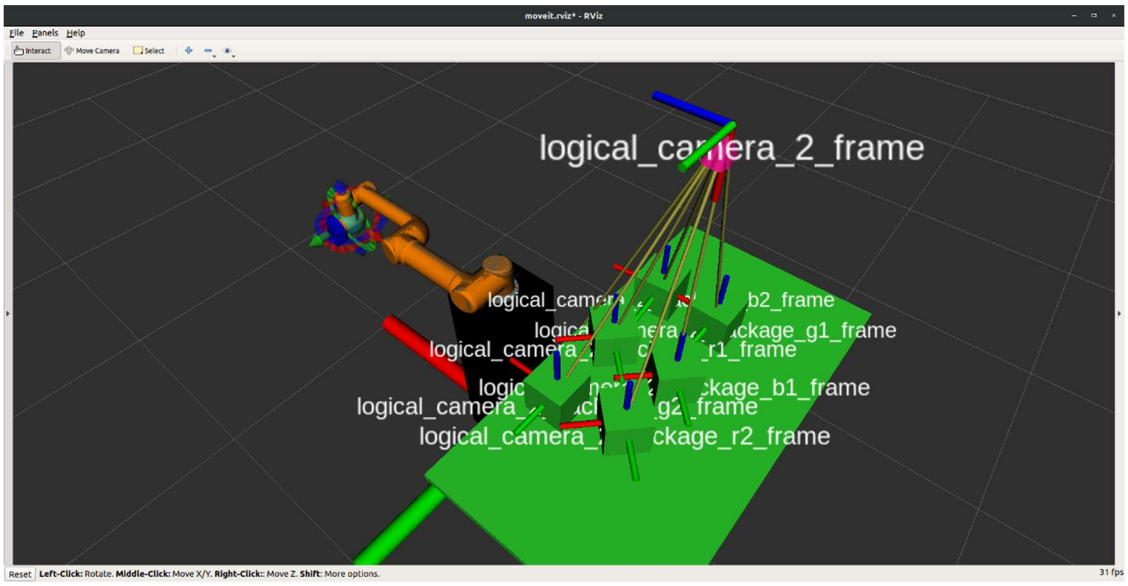


Fig. 4: The TF frame visualisation of the boxes in RViz.

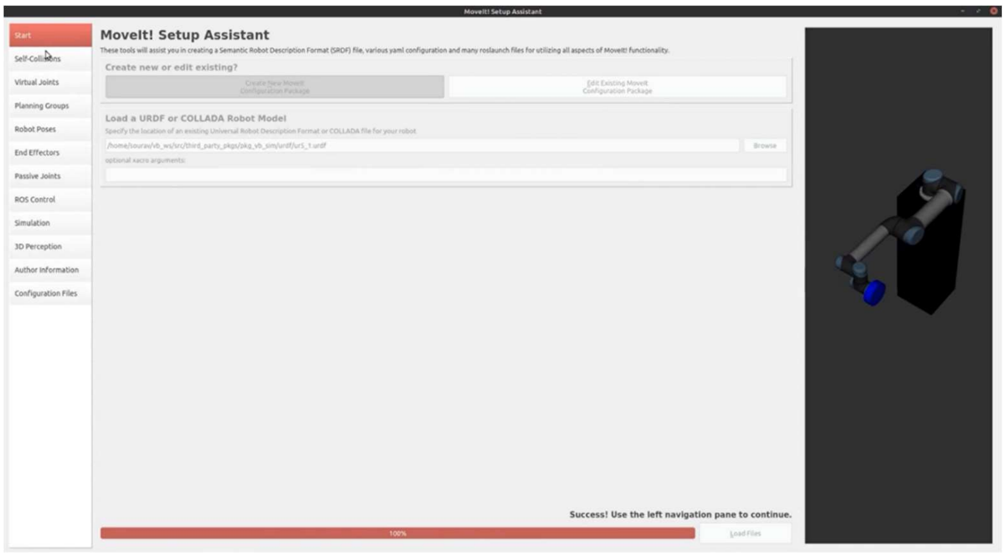
Setting up MoveIt!

To create the MoveIt! controllers for the UR5, the MoveIt! Setup Assistant [13] helps create a controller and other necessary configuration files for use with the MoveIt! pipeline for the UR5 based off its URDF, as shown in Fig. 5. Through the Setup Assistant, the active joints directly controlled by the controller, passive joints like the gripper link and base-ground link, and the end-effector links are defined, as well as the motion planning algorithms to be selected to be used whilst executing motion planning algorithms.

However, the default MoveIt! controllers did not interface properly with the UR5, and required further tweaking to conform to the controller used for this implementation, namely the *FollowJointTrajectory* type. The configuration file for the controller is as below in Snip. 1:

```
controller_manager_ns: /
controller_list:
- name: ur5_1_controller
action_ns: follow_joint_trajectory
type: FollowJointTrajectory
joints:
- ur5_shoulder_pan_joint
- ur5_shoulder_lift_joint
- ur5_elbow_joint
- ur5_wrist_1_joint
- ur5_wrist_2_joint
- ur5_wrist_3_joint
```

Snip. 1: The custom configuration file for the MoveIt! controller.



MoveIt! Setup Assistant

These tools will assist you in creating a Semantic Robot Description Format (SRDF) file, various yaml configuration and many roslaunch files for utilizing all aspects of MoveIt! functionality.

Create new or edit existing?

Load a URDF or COLLADA Robot Model

Specify the location of an existing Universal Robot Description Format or COLLADA file for your robot

/home/sourav/vb_ws/src/third_party_pkgs/pkg_vb_sim/urdf/ur5_1.urdf

optional xacro arguments:

Success! Use the left navigation pane to continue.

Fig. 5: The MoveIt! Setup Assistant, which creates the controller package for the UR5.

Total Integration with ROS Scripting

The ROS scripts are responsible for tying all these disparate elements together. The main ROS-Launch file [10] that handles the execution of all assets is as shown in Snip. 2:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    Launch file for the task.
 -->
<launch>

    <!-- Launch Simulation Environment in Gazebo -->
    <include file="$(find env_sim)/launch/gazebo_task_simulation.launch" />

    <!-- Launch Moveit move_group Node -->
    <include file = "$(find pkg_moveit_ur5_1)/launch/move_group.launch" />

    <!-- Run RViz with Moveit! Plugin -->
    <arg name="use_rviz" default="true" />

    <include file="$(find pkg_moveit_ur5_1)/launch/moveit_rviz.launch" if="$(arg use_rviz)">
        <arg name="rviz_config" value="$(find pkg_moveit_ur5_1)/launch/moveit.rviz"/>
    </include>
    <!-- Spawn boxes on the table -->
    <node name= "gazebo_spawn_models" pkg= "env_sim" type="gazebo_spawn_models.py"/>

    <!-- Add objects to Moveit! Planning Scene in RViz -->
    <arg name="scene_file" default="$(find ur5_control)/config/rviz/task.scene"/>
    <node name = "moveit_publish_scene_from_text" pkg= "moveit_ros_planning" type =
"moveit_publish_scene_from_text" args= "$(arg scene_file)"/>

    <!-- Run pick-place node -->
    <node name= "node_ur5_sorter" pkg= "ur5_control" type="ur5_sorter.py" output="screen"/>

</launch>
```

Snip 2: The ROS-Launch file.

The actual control is handled by `ur5_sorter.py`, which utilises `lib.py`, a class object that contains the essential functions of MoveIt! and Rviz handled by ROS. The entire has been elaborated in detail [10] in Snip. 3:

```
#! usr/bin/env python
'''
Library of UR5-specific functions for MoveIt! and RViz Motion Planning.
Contains:
1) MoveIt! Parameters and Controllers for controlling the arm
    a) Go to a specified pose
    b) Go to specified join angles
2) RViz Planning Scene controls for adding, attaching, detaching and removing objects from
the Planning Scene
'''
import rospy, sys, moveit_commander, moveit_msgs.msg, actionlib, tf

class UR5MoveIt:
    '''
    Class object for the UR5 arm.
    '''
    def __init__(self):
        '''
        Constructor containing all essential assets for MoveIt! and RViz.
        '''
        # Initialise the Node
        rospy.init_node('node_ur5', anonymous=True)

        # Transform Listener for box TF detection
        self.tf_listener = tf.TransformListener()

        # RViz and MoveIt parameters
        self._box_name = "box"
        self._planning_group = "ur5_1_planning_group"
        self._commander = moveit_commander.roscpp_initialize(sys.argv)
        self._robot = moveit_commander.RobotCommander()
        self._scene = moveit_commander.PlanningSceneInterface()
        self._group = moveit_commander.MoveGroupCommander(self._planning_group)
        self._display_trajectory_publisher = rospy.Publisher(
            '/move_group/display_planned_path', moveit_msgs.msg.DisplayTrajectory,
queue_size=1)

        self._execute_trajectory_client = actionlib.SimpleActionClient(
            'execute_trajectory', moveit_msgs.msg.ExecuteTrajectoryAction)
        self._execute_trajectory_client.wait_for_server()

        self._planning_frame = self._group.get_planning_frame()
        self._eef_link = self._group.get_end_effector_link()
        self._group_names = self._robot.get_group_names()
        self._touch_links = self._robot.get_link_names(group=self._planning_group)

        rospy.loginfo('\033[94m' + "Planning Group: {}".format(self._planning_frame) +
        '\033[0m')
        rospy.loginfo(
            '\033[94m' + "End Effector Link: {}".format(self._eef_link) + '\033[0m')
        rospy.loginfo(
            '\033[94m' + "Group Names: {}".format(self._group_names) + '\033[0m')
        rospy.loginfo('\033[94m' + " >>> Ur5Moveit init done." + '\033[0m')

    def set_joint_angles(self, arg_list_joint_angles):
        '''
        Goes to specified joint angles.
        Parameters:
             arg_list_joint_angles (float[]): A list of joint angles in radians to plan
             towards.
        Returns:
             flag_plan (bool): Confirmation whether the planning and execution was successful
             or not.
        '''

        list_joint_values = self._group.get_current_joint_values()
        rospy.loginfo('\033[94m' + ">>> Current Joint Values:" + '\033[0m')
        rospy.loginfo(list_joint_values)

        self._group.set_joint_value_target(arg_list_joint_angles)
        self._group.plan()
        flag_plan = self._group.go(wait=True)

        list_joint_values = self._group.get_current_joint_values()
        rospy.loginfo('\033[94m' + ">>> Final Joint Values:" + '\033[0m')
        rospy.loginfo(list_joint_values)

        pose_values = self._group.get_current_pose().pose
        rospy.loginfo('\033[94m' + ">>> Final Pose:" + '\033[0m')
        rospy.loginfo(pose_values)

        if (flag_plan == True):
            rospy.loginfo('\033[94m' + ">>> set_joint_angles() Success" + '\033[0m')
        else:
            rospy.logerr('\033[94m' + ">>> set_joint_angles() Failed." + '\033[0m')

        return flag_plan


    def add_box(self, box_name, box_length, box_pose):
        '''
        Adds a box to the RViz planning scene.
        Parameters:
            box_name (str): The name to be assigned to the box.
            box_length (float): The size of the box.
            box_pose (PoseStamped object): The pose and orientation of the box.
        '''
        self._scene.add_box(box_name, box_pose, size=(box_length, box_length, box_length))

    def attach_box(self, box_name):
        '''
        Attaches the specified object(box) to the robot hand.
        Parameters:
            box_name (str): The name of the box in the RViz Planning Scene te be attached.
        '''
        self._scene.attach_box(self._eef_link, box_name, touch_links=self._touch_links)
    def remove_box(self, box_name):
        '''
        Removes the specified object(box) from the RViz Planning Scene.
        Parameters:
            box_name (str): The name of the box to be removed.
        '''
        self._scene.remove_world_object(box_name)

    def __del__(self):
        '''
        Destructor for the class object.
        '''
        moveit_commander.roscpp_shutdown()
        rospy.loginfo(
            '\033[94m' + "Object of class Ur5Moveit Deleted." + '\033[0m')
```

Snip. 3: The class object containing essential MoveIt! and RViz assets.

Output

The total setup [14] in Fig. 6 and 7 demonstrates the whole operation in action.

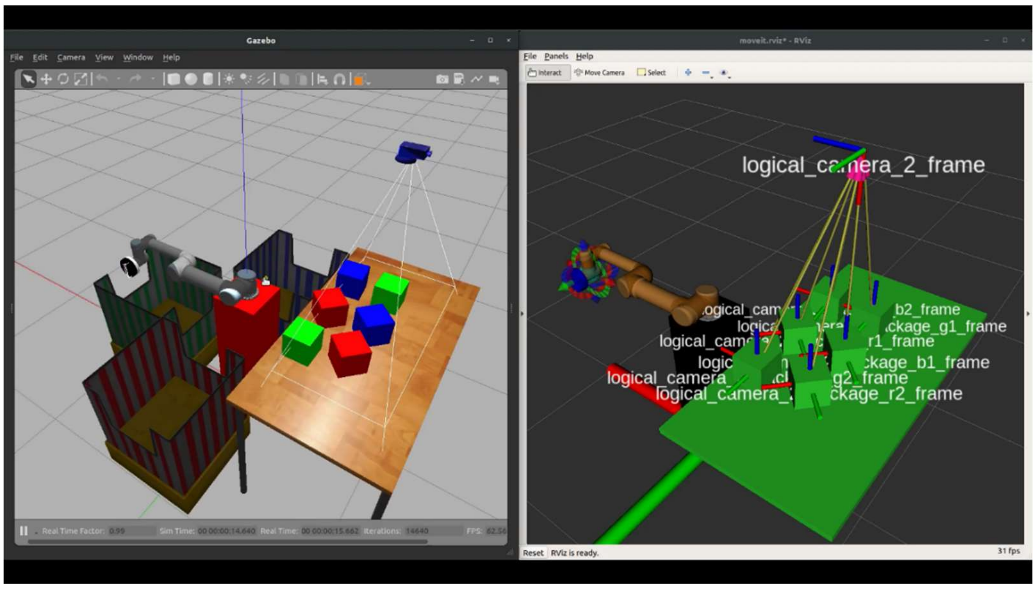


Fig. 6: The whole simulation setup combining Gazebo, MoveIt!, RViz and ROS.

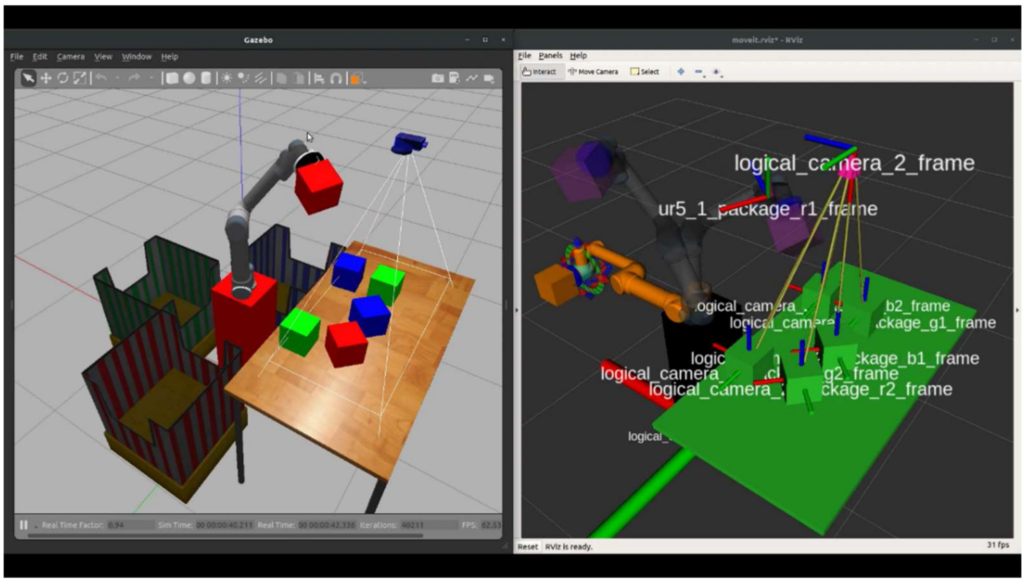


Fig. 7: The UR5 picking up a box.

Result Analysis and Discussion

In the course of simulation, the following points of discussion pertaining to the simulation arose:

Logical Camera and TF Updation

The TFs visualised in the setup have a default timeout rate of 15 seconds [15]: that is, when a box (and by extension, its TF) travels out of the detectable range of the Logical Camera, the frames linger for a few seconds before fading. This poses problems as the camera assumes the TF still exists, leading the UR5 to travel to non-existent positions. This is rectified by changing the Frame Timeout in the RViz interface from the default 15 seconds to 1 second.

Vacuum Gripper

The vacuum gripper on the UR5 in Fig. 8 consists of a logical camera which indicates whether the object in its vicinity is pickable or not. One problem that arose was the unintended attachment of object in close vicinity to the gripper. This can be potentially fixed by tweaking the link-attach function to make it more concise, or eradicating the logical camera [3] attached to the gripper altogether as the messages posted by it may have unintentionally interfered with the working.

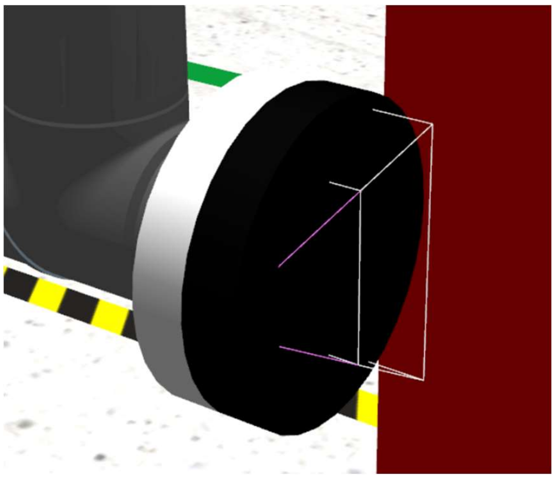


Fig. 8: The vacuum gripper on the UR5.

Conclusion

The report demonstrates a preliminary setup for parameterised sorting of objects using a robotic manipulator. The specific parameters to be considered whilst sorting as well as the manipulator to be used may differ between applications as elaborated previously. The various applications derived from this setup were also discussed.

The speed of the implementation may be further improved by removing the requirement of traversing the arm to a home position as long as there are objects detected by the logical camera. Implementing more compact code will also contribute to lightness of code, which further improves execution by alleviating the stress on hardware resources.

Future Work

The concepts in this implementation may be applied towards more advanced applications like sorting based on shape, or sorting based on size. In each case the parameters may be switched over or even combined depending on the application as mentioned in Table 1.

References

[1] M. Sami Fadali, Antonio Visioli, "Introduction to Digital Control", *ScienceDirect* [Online]. https://www.sciencedirect.com/topics/engineering/robotic-manipulator

[2] Eric Esson, "How Industrial Standards for Robotics Increase Safety and Efficiency", *ISA Interchange* [Online]. https://blog.isa.org/new-industrial-standards-robotic-safety

[3] Open Source Robotics Foundation, "Logical Camera Sensor", *Gazebo* [Online]. http://gazebosim.org/tutorials?tut=logical_camera_sensor&cat=sensors.

[4] Open Source Robotics Foundation, "Gazebo", [Online]. http://gazebosim.org/.

[5] Open Source Robotics Foundation, "rviz", *ROS Wiki* [Online]. http://wiki.ros.org/rviz.

[6] PickNik Robotics, "MoveIt", [Online]. https://moveit.ros.org.

[7] Open Source Robotics Foundation, "urdf", *ROS Wiki* [Online]. http://wiki.ros.org/urdf.

[8] Open Source Robotics Foundation (2020), *gazebo_models* [GitHub]. https://github.com/osrf/gazebo_models.

[9] Universal Robots (2020), *universal_robot* [GitHub]. https://github.com/ros-industrial/universal_robot.

[10] Neehal Sharrma (2020), *MTE-Robotics-Lab* [GitHub]. https://www.github.com/CH13F-1419/MTE-Robotics-Lab.

[11] Sammy Pfeiffer (2016), *gazebo_ros_link_attacher* [GitHub]. https://github.com/pal-robotics/gazebo_ros_link_attacher.

[12] Open Source Robotics Foundation, "Gazebo ROS API for C-Turtle", *ROS Wiki* [Online]. http://wiki.ros.org/simulator_gazebo/Tutorials/Gazebo_ROS_API.

[13] PickNik Robotics, "MoveIt! Setup Assistant", *MoveIt! Tutorials* [Online]. https://ros-planning.github.io/moveit_tutorials/doc/setup_assistant/setup_assistant_tutorial.html.

[14] Neehal Sharrma (2020), "Pick-place sorting with a UR5", *YouTube* [Online]. https://youtu.be/oqL3__f0exw.

[15] Open Source Robotics Foundation, "tf", *ROS Wiki* [Online]. http://wiki.ros.org/tf.