ECE 470- Project Report

Team Name: SplitBot

Members :

1. Amith Ramdas Achari – amrithr3@illinois.edu
2. Raatan Venkataraman Subashini – raatanv2@illinois.edu

GitHub link: https://github.com/amithachari/ur5_ROS-Gazebo

YouTube Simulation link: Pick and Place Robot - Warehouse Automation

1. Introduction

In the recent days, with the increase in online orders and shipping to the customers, warehouses play an important role as sorting and storage locations. The number of packages being ordered online is skyrocketing, pushing the demand for the need of quicker delivery options. In a package delivery pipeline, warehouses serve as temporary storage location until the package is assigned to an appropriate delivery channel. Warehouses receive a large number of packages in a day, which needs to be sorted and delivered to various locations. This poses a serious problem as employing a large number of people for this mundane task would drive the cost of the business to a great extent. Keeping all these factors in mind, we could conclude that automating the sorting process would potentially save a lot of time and money.

Splitbot is an efficient sorting solution which can be employed in warehouses to sort boxes quickly based on physical attributes like color, weight, shape etc. The goal of this project is to complete a pick and place task by segregating red and blue boxes into appropriate storage stations in a warehouse setting. The simulation is carried out on gazebo with the help of available packages and tailor them to this application. The kinematics of the robot was entirely taken care by MoveIt, which made it possible to complete the pick and place without halting the conveyer.

2. Experimental Setup and Methodology

2.1 Task Description

When the boxes enter the field of view of the camera sensor, the centroid and the color of the box are determined. When the box enters the task space, it is picked up on the go and dropped onto the appropriate storage location. The process is briefly outlined in the Figure 1.

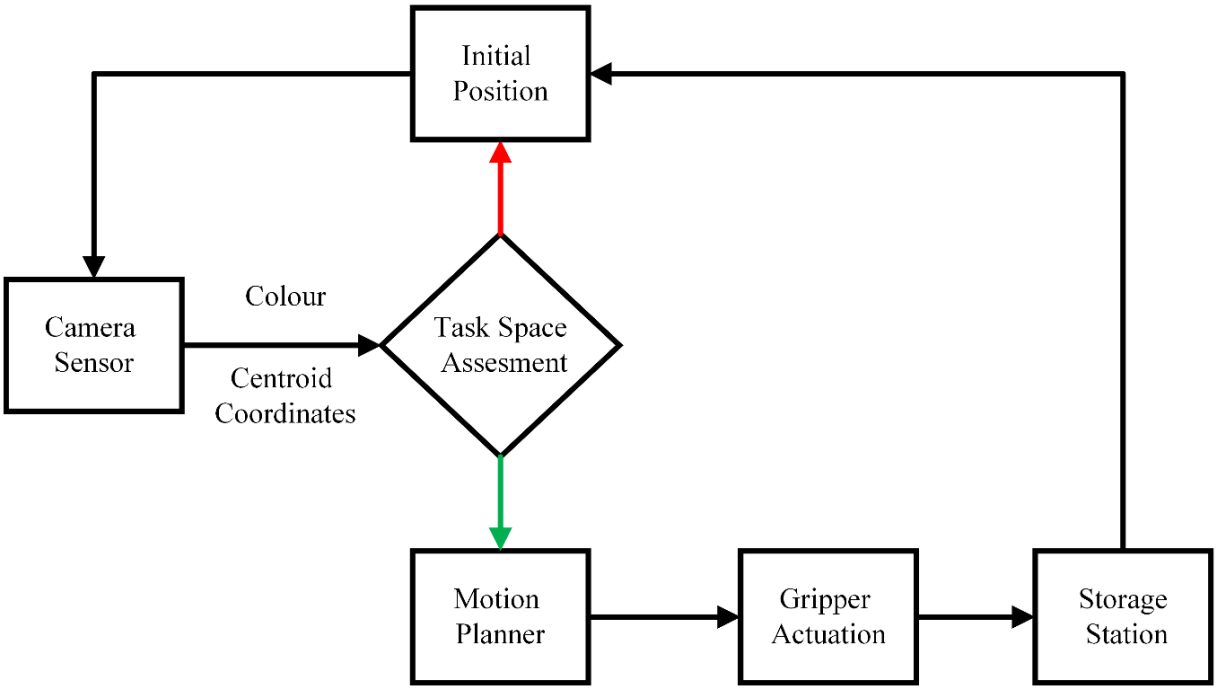


Figure 1 : Pictorial description of processes outline

2.2 Robot Description:

In this project, we have used UR5 to be our robot manipulator for completing this pick and place task. It is a 6 joint robot with each joint having a 6 degree of freedom. Figure 2(a) describes zero position of the robot with the all the joint axis marked. Figure 2(b) describes

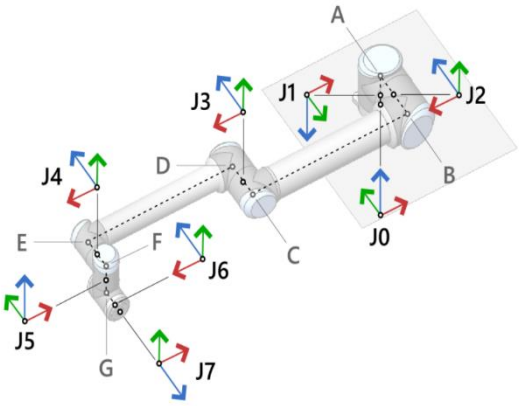


Figure 2(a): UR5 at Zero Configuration.

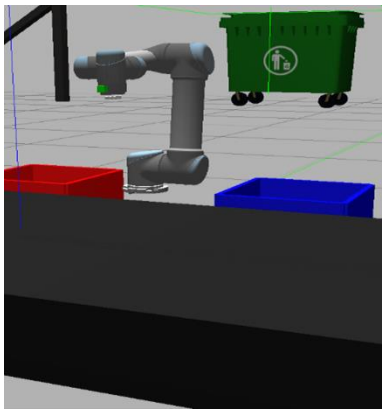


Figure 2(b): UR5 at Initial Position.

the initial location of the robot at the start of the simulation. UR5 is an ideal robot for simple and lightweight pick and place application. Since we had an opportunity to build our fundamentals and understanding of robot using UR3 in the lab sessions, we wanted to explore the working of UR5 as well.

2.3 Camera Sensor:

The camera sensor is one of the key components of this robot for identifying the color and giving feedback to the motion planner. For the camera, we used a gazebo plugin camera, which was then added onto the URDF file. The camera used was *usb_cam*, which was able to detect the blocks. It is through a dedicated node called *usb_cam* node, the information of image is published. Image published out of the vision node is processed in the *vision_node* to identify the color and publish the centroids of the identified block.

The camera sensor used in this project is similar to an USB camera hardware. The communication in and out of the sensor is established through a *usb_cam_node()*. The camera node publishes the output image as a topic given by: *<camera_name>/image_raw(sensor_msgs/Image)* and has a key set of parameters listed as follows.

1. *~video_device (string, default: "/dev/video0")* - Camera on/off signal
2. *~image_width (integer, default: 640)*
3. *~image_height (integer, default: 480)*
4. *~pixel_format (string, default: "mjpeg")*
5. *~io_method (string, default: "mmap")*

Once the image is published out of the *usb_cam* node, it is processed using the *vision_node*. Identification of the color of the block is done as outlined in the following steps

1. Converting the obtained image into HSV format using the function *cv2.cvtColor()* which takes in two parameters. The first being image and second is *cv2.COLOR_BGR2HSV*.
2. Masked Image is obtained by using the function *cv2.inRange(Image_hsv, lower, upper)*. The function masks out the colors in range specified as input parameters lower and upper mentioned in the function definition.

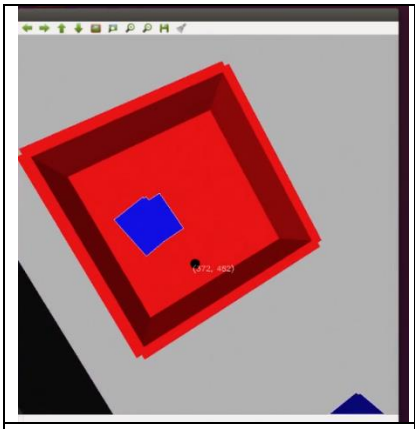
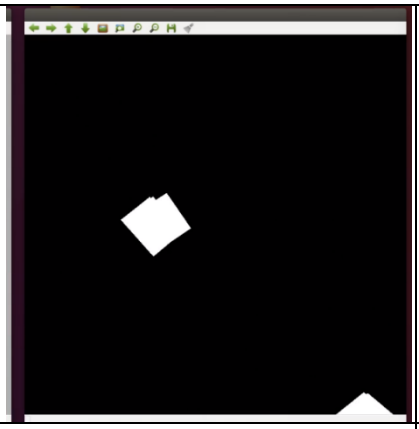


Figure 3(a) Raw Image



Figrue 3(b) Blue body Mask

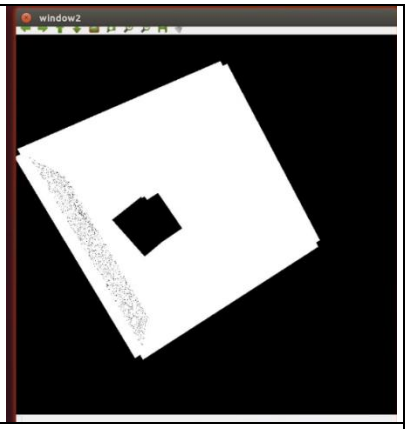


Figure 3(c) Red body Mask

In the above figure, Figure 3(a) is published out of the vision node. On entering the CV node, the image is first converted into HSV image. Figure 3(b) is obtained through applying a filter in which the pixel values on the HSV image range between ([100,150,0]) and ([140,255,255]). In such case, the objects which are originally blue will be masked out and rest of the image will be black. It is very much like filtering out the blue pixels out of the image. Similarly, when the HSV range is between ([0, 100, 100]) and ([140,255,255]), the red colored pixels in the image are masked out, similar to what is shown in Figure 3(c).

After the masked images are obtained, the centroids are calculated using the bounding rectangle method. Since the masked image contains the edges of the rectangle, the midpoints can be easily located by dividing length and height by a factor of two.

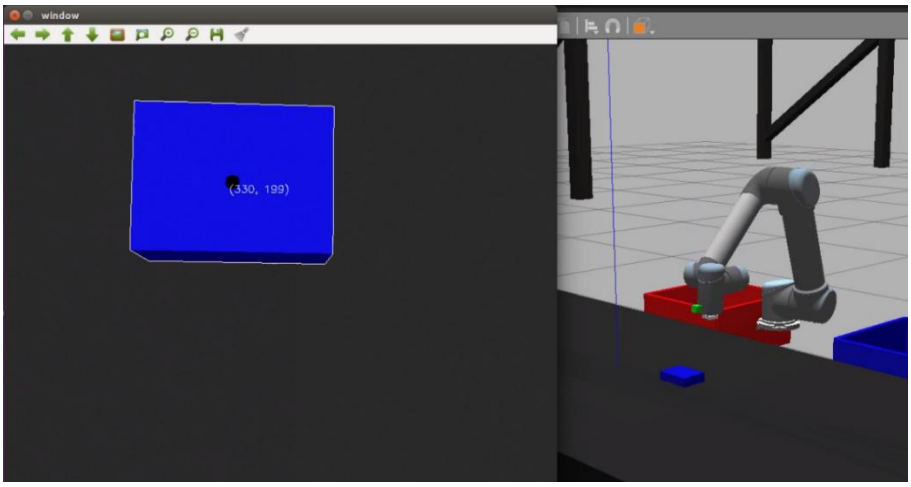


Figure 4: Locating the centroid of the identified block

2.4 End Effector/Gripper:

The end effector is also modified to increase the payload capacity of the robot. Instead of one suction cup, we used nine suction cups arranged in an array of three rows and three columns as shown in Figure 5. The suction pressure applied onto the cups are equal and hence will ensure proper grip onto the block.

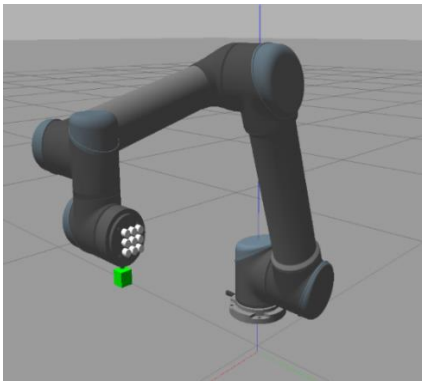


Figure 5: View of the set of grippers appended on to the robot.

2.5 Scene Description:

The conveyer is built on the interface by modifying the URDF file of the environment. At one end of the conveyer, blue and red blocks are spawned. The user input is taken for spawning the blocks as blue and red. The world/surrounding is created using a world file which is written to spawn a working environment like a warehouse. Some components of the world file were taken from *aws-robotics/aws-robomaker-small-house-world*.

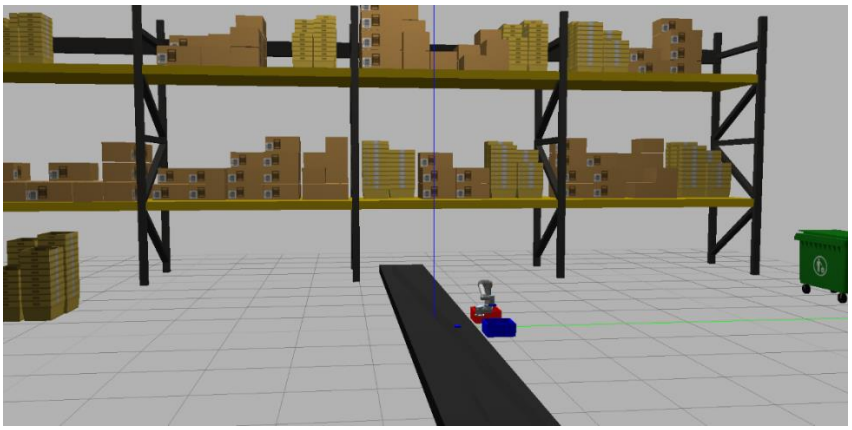


Figure 6: Warehouse Environment

3. Manipulation

Once, the image from the camera is taken from the *usb_camera*, it publishes the centroid location. So, the next phase it to manipulate our robot to the block and turn on the gripper. This is done with the help of MoveIt. The configuration of the modified URDF file is done in the Setup Assistant offered by MoveIt. This configured file can then be used to simulate the robot dynamically. The setup assistant can be launched by typing *roslaunch moveit_setup_assistant setup_assistant.launch*. From here we can create a new Configuration Package. The setup_assistant lists out a set of panels to setup our configuration file as follows:

1. **Self-Collision:** This allows us to check a number of possible self-collisions which can occur between the links of the manipulator itself. In our case, we used the default level of sampling and created the *Collision Matrix.*
2. **Virtual Joints:** This is useful to link the robot with the scene, so we created a simple virtual joint, called the world, with a parent called world.
3. **Planning groups:** This is the most important section of this setup, this part connects all the links and joints, which can further be used for kinematics. We specified which joints compose the arm and which joints composed of the gripper.
4. **Robot Poses:** This allowed us to define some of the initial poses for the robot. So, we declared our initial poses and two end poses in this part of the setup.
5. **End Effector:** We had to explicitly define which part of the planning group was our end effector, and in our case, it was the gripper.
6. **Author Information:** Information of the author was provided.
7. **Configuration file:** This allowed us to generate our new configuration file

Figure 7: Defining planning groups

3

Figure 8: Self collision check

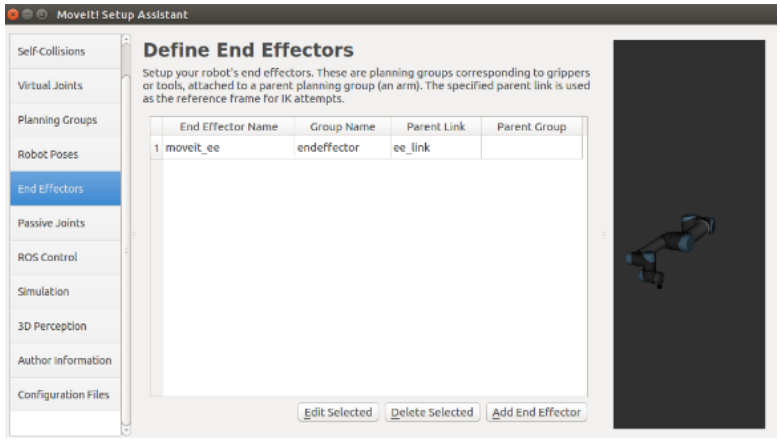


Figure 9: Defining end effectors

3.1 Motion Planning Node

move_group is a ROS node. It basically uses the ROS param server to get three different kinds of information namely: URDF, SRDF and MoveIt Configuration file which we just generated using the setup assistant. Move_group will look for configuration specific to MoveIt like joint limits, kinematics, motion planning, perception, and other information. The configuration file for these components is automatically generated by the MoveIt setup assistant and stored in the desired config directory. The MoveIt node was launched by calling the move_group.launch in the launch file UR5_initialize.launch.

3.2 Robot Interface

move_group communicates with the robot through ROS topics and actions. It gets the current state like position, velocity of the joints using this mode of communication, and then communicates the processed information to the robot controllers.

3.3 Motion Planning

MoveIt can use motion planners by adding it as a plugin interface. This enables MoveIt to communicate with different motion planners. The interfacing between each of these motion planners is done through ROS actions and services. In our case we used the default motion planner offered by *move_group* which is the OMPL, and interfacing was done through MoveIt setup assistant.

Figure 10: Trajectory of the robot passing through waypoints specified by the planner

3.4 Motion Plan Request

Once the setup is done, we need MoveIt to generate a result for us, and this can be done after requesting what the motion planner would like to do. In our case, we want the motion planner to move our arm to a different location in the joint space or the end effector to a new pose. There won't be any collisions with the robot as the collision matrix was generated in the setup. There can be constraints specified while passing the request, usually the constraints are kinematic constraints such as position constraints, orientation constraints, joints constraints, velocity constraints or acceleration constraints.

3.5 MoveIt Motion Plan Result

The *move_group* node then generates the desired trajectory required in response to the motion plan request. This trajectory generated is not a path and is a trajectory, which means that it will use the constraints like maximum velocities, etc. to generate a trajectory which obeys these constraints.

3.6 Kinematics

MoveIt also contains a default inverse kinematics plugin namely the KDL numerical Jacobian-based solver. However, one can use a custom inverse kinematics algorithm and use it as a plugin. The default plugin is automatically configured in the configuration file generated by the MoveIt setup assistant.

3.7 UR5_Planning Node

The planning for the system was done through the planning node, namely *ur5_mp.py*. This node essentially planned the waypoints for the robot and published flags for the gripper node, to either turn on or turn off the gripper. The planning node subscribed to the message published by the camera, which is centroid position of the block. This then tracked the block based on the centroid position and as soon as the gripper was close to the block it published True flag to the gripper node, and hence turning on the gripper. The *move_group* was initialized using the *moveit_commander.MoveGroupCommander('manipulator')*. And then *moveit_group* planned the trajectory accordingly.

So, it computed a sequence of waypoints that could make the end-effector move in a straight-line segment, that followed the position specified at each waypoints. The configurations were computed at every small step, 0.01m. The return value of this step is a tuple: which tells the fraction of how much the path was followed corresponding to the actual robot trajectory.

3.8 Gripper Node:

The actuation of this gripper to hold the object of interest is achieved by calling the gripper node. Gripper receives input from the motion planner whenever the detected block enters the task space. The gripper_node keeps subscribing to the message published by vision_node, and the actuation of the gripper is done based on the flag in the message.

When there is an input from the motion planner, the gripper status is changed to *True* by using the turn on function. Once the gripper is turned on, gripper subscribes its messages, and then *gripper_callback()* gets activated.

4. Data and Results:

The communication between the robot_state_publisher and Gazebo is done through */joint_states* topic as shown in the Figure 11. The *joint_state* contains different messages namely header, name (name of each joint), position, velocity, and effort. We plotted the real time data of the *joint_states* using *rqt_graph*. *rqt_graph* creates a dynamic graph of what's going on in the system. *rqt_graph* is part of the rqt package, which enables us to plot the position vs time for each joint and velocity vs time for each joint. The trajectory generated is the result of MoveIt, which was generated once the inputs were fed in through *moveit_command* it returns the planned trajectory consisting of waypoints, position, velocity, and acceleration obeying the limits.

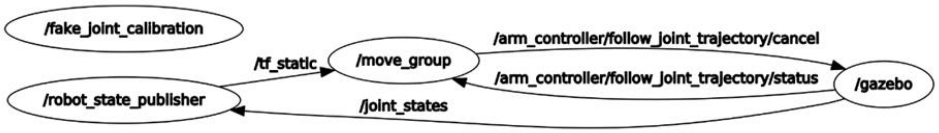


Figure 11 : RQT Graph describing communication between robot_state_publisher and gazebo

The joints are named as [elbow_joint, shoulder_lift_joint, shoulder_pan_joint, writst_1_joint, wrist_2_joint, wrist_3_joint].

Note that the following plots are for a time interval where the block comes in contact with the gripper during the simulation of Splitbot.

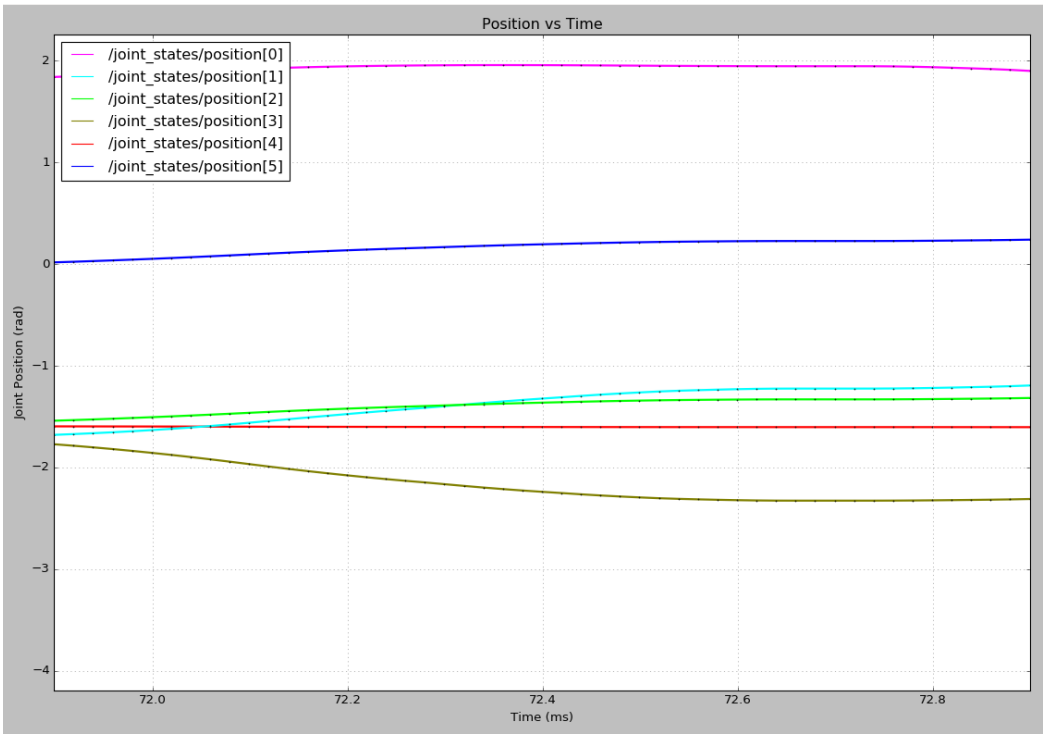


Figure: rqt_plot represents position of each joint vs time

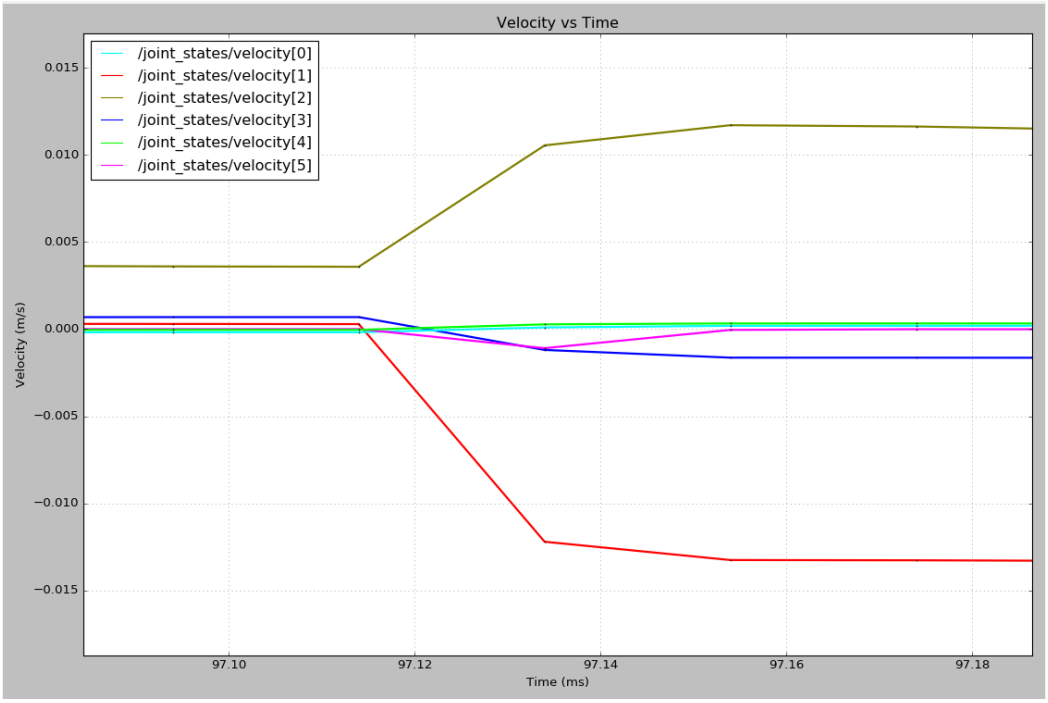
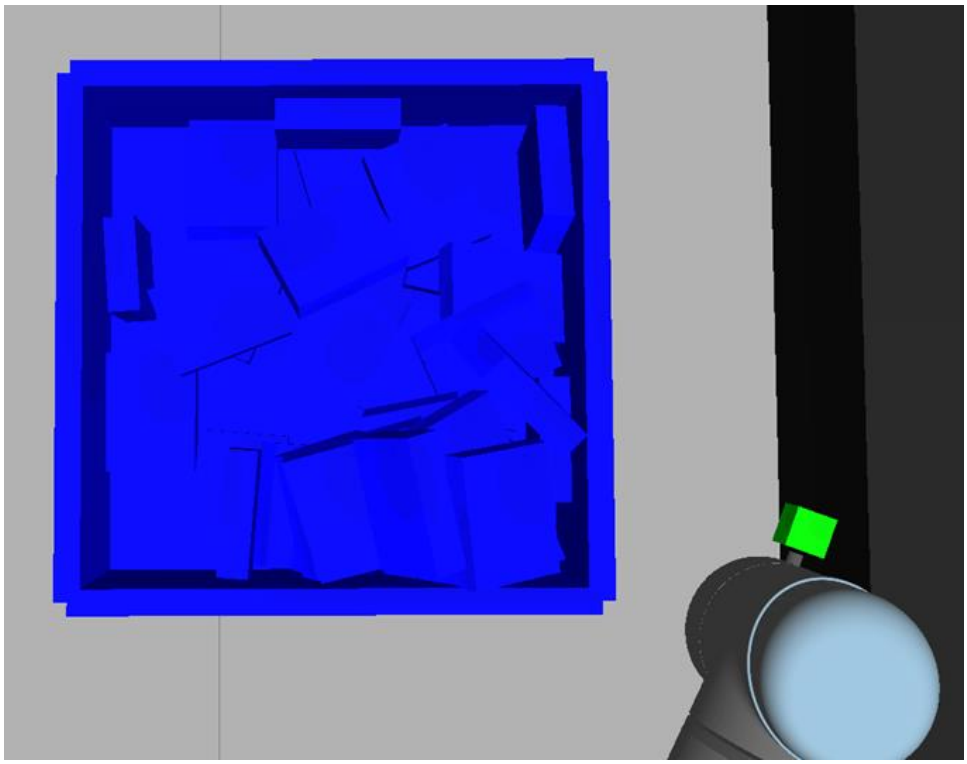


Figure: rqt_plot represents position of each joint vs time

5.1 Success Rate/Failure Case



The entire simulation was done in VMWare Workstation 16, with Ubuntu 16.04 and ROS Kinetic. The specification of the machine being 16GB RAM, 6GB Nvidia Graphics Card 3060Ti, and Intel i7 11th gen processor. This simulation was conducted for 10 mins. And a total of 37 blue blocks were spawned in this time. And the robot was able to pick all the 37 blocks and sort them accordingly. A similar test was done for red blocks, and it was able to sort all 37 of the blocks.

Color	Number of Trials	Number of Success	Success Percentage
Blue	37	37	100
Red	37	37	100

6. Summary and Challenges:

1. Constructing the URDF File:

During the construction of the URDF File, apart from appending the grippers and the camera plugins from gazebo, the linking of the camera to the object of interest to be detected was a challenging task.

2. Object detection:

On utilizing the camera sensor to detect the blocks and their centroids. We face a tricky hurdle while detecting the color of two different blocks. For instance, when the blue block is conveyed, on successfully detecting the blue block, the following red block was not detected and vice versa. We overcame this challenge by using a set of two masks for the two different colored blocks conveyed on the conveyer.

3. Learning to play with MoveIt:

While working around gazebo with MoveIt and the camera, the major hurdle was to publish the centroid points detected by the camera to *ur5_mp.py* node. We were also facing difficulty in subscribing multiple messages in the *ur5_mp.py node*. With the help of MoveIt docs and tutorials, we were able to work through it.

7. Conclusion

The robot was able to detect the color 100% of the times. Further, it was able to locate centroid and successfully pick the block at each iteration. Instead of defining the inverse kinematics, we made use of the motion planning library which calculates waypoints and joint angles for the given task. Once the camera publishes images through *usb_cam* node, which the *vision* node takes in and publishes CV coordinates. These coordinates are fed into the planner for calculating waypoints and respective joint angles for successful actuation. We encountered several challenges while interfacing the camera and motion to get proper solution for the inverse kinematics.

The knowledge we gained from laboratory sessions in using ROS nodes for forward kinematics, inverse kinematics and computer vision was instrumental in completing this task on the simulation environment.

8. Further Improvements/Recommendations

- Since this version of Splitbot is stationary pick and place sorting bot, we will be working on a mobile version of it, which can navigate to different sorting locations after picking the boxes from a conveyor.
- We also would like to extend this project by coupling the robot motion planning with a bar code scanner, process the delivery package information and sort based on other attributes like location, size and the type of package being delivered.

9. References

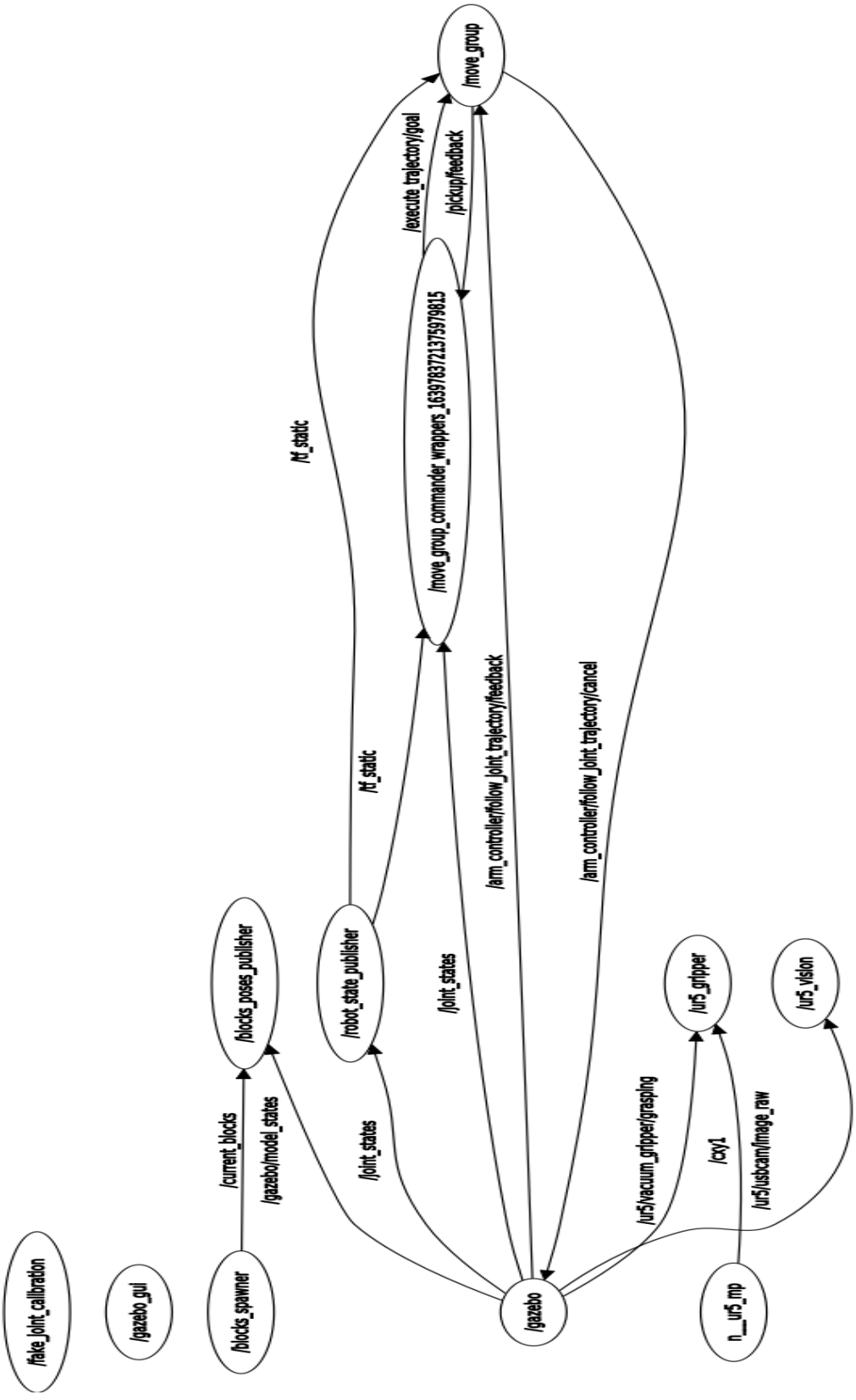
https://wiki.ros.org/ROS/Tutorials

https://moveit.ros.org/documentation/concepts/

https://industrial-training-master.readthedocs.io/en/melodic/_source/session6/Using-rqt-tools-for-analysis.html

Huang, L., Zhao, H., Implementation of UR5 pick and place in ROS-Gazebo with a USB cam and vacuum grippers, (2018), GitHub repository, https://github.com/lihuang3/ur5_ROS-Gazebo.git

Appendix


/move_group
/execute_trajectory/goal
/pickup/feedback
/move_group_commander_wrappers_1639783721375979815
/tf_static
/tf_static
/robot_state_publisher
/joint_states
/arm_controller/follow_joint_trajectory/feedback
/arm_controller/follow_joint_trajectory/cancel
/blocks_poses_publisher
/current_blocks
/gazebo/model_states
/joint_states
/ur5/vacuum_gripper/grasping
/ur5_gripper
/cxy1
/ur5/usbcam/image_raw
/ur5_vision
/make_joint_calibration
/gazebo_gui
/blocks_spawner
/gazebo
n__ur5_mp