Cube Sorting Manipulator

MEAM 520: Intro to Robotics

Final Report

Abhishek Patil
Nagarakshith Makam Sreenivasulu

December 11th 2019
Instructor: Cynthia Sung

Contents

Problem Statement

The goal of this project is to use the Lynx manipulator to pick and place cubes of known dimensions in a pre-defined obstacle space.

Motivation

The primary focus was on learning pertinent tools like the Robot Operating System (ROS). We also wanted to work on a project wherein we could use all the concepts that we learnt in class during this course, namely: Transformations, Forward and Inverse Kinematics, Path Planning, and Sensing. This work is an amalgamation of all these components

Approach

To solve the problem as described our system can be bifurcated into three modules- Sensing, Planning and Execution. ROS is the framework which integrates all these modules.

Robot Operating System

ROS is an open-source, meta-operating system for the robot. It provides services like hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

Sensing

To localize the cubes present in the arena we chose the approach of using the overhead camera and the Aruco markers. Every cube has an aruco marker attached to it.

An Aruco marker is a synthetic square marker composed by a wide black border and an inner binary matrix which determines its identifier (ID). Every aruco marker has a size which represents the number of bits it contains. And in turn, the aruco marker is derived from a dictionary of a specified size (the number of markers in that dictionary).

From the image obtained using the overhead camera, the aruco markers are detected. Firstly, contours are extracted from the image and those which approximate to the square shape and are convex are considered the potential marker candidates. Using some computer vision techniques these candidates are filtered to obtain the actual markers and are decodified to obtain their ID.

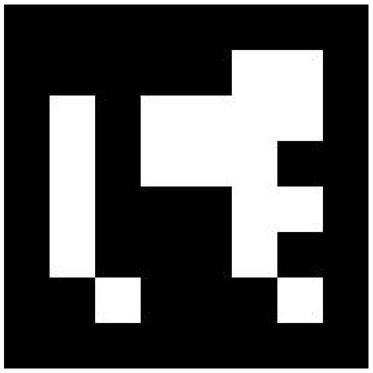


Fig.1 Aruco Marker

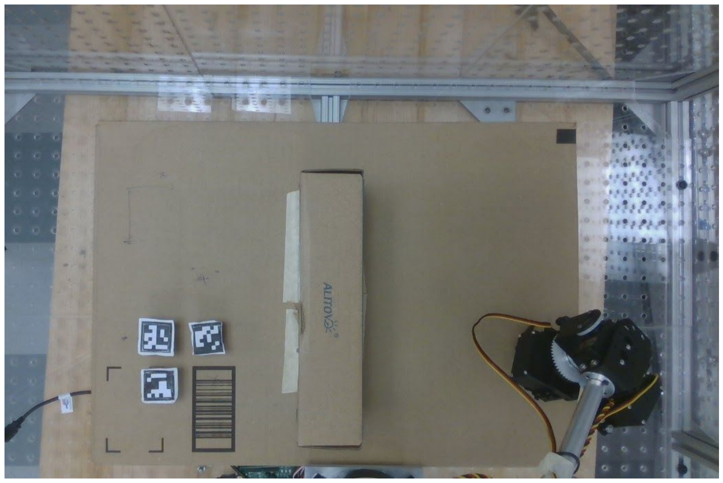
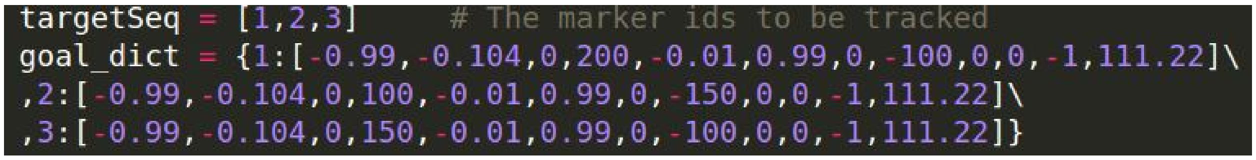


Fig.2 Camera View

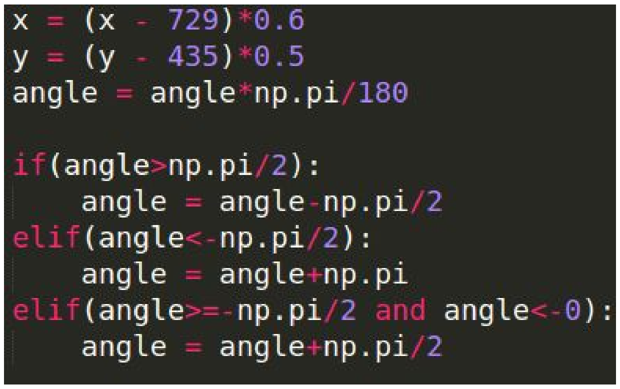
OpenCV has the package to detect the aruco markers. The aruco markers from the dictionary DICT_6X6_250 of OpenCV are used in this work. We created the 'aruco_ros_opencv' package to identify the specified aruco markers and publish the information on the topic '/aruco/aruco_info_out'.
Also, the image with the outlined aruco markers drawn is published on the topic '/aruco/marked_image_out' which can be visualized using the image_view ROS package.

The markers to be identified are to be defined in the targetSeq list. This is the order in which the cubes are picked. Also, the goal transformation matrix associated with each marker is to be defined in the goal_dict of the 'aruco.py' script.


```
targetSeq = [1,2,3]      # The marker ids to be tracked
goal_dict = {1:[-0.99,-0.104,0,200,-0.01,0.99,0,-100,0,0,-1,111.22]\
,2:[-0.99,-0.104,0,100,-0.01,0.99,0,-150,0,0,-1,111.22]\
,3:[-0.99,-0.104,0,150,-0.01,0.99,0,-100,0,0,-1,111.22]}
```
Fig. 4

The camera calibration was done to obtain the extrinsic camera parameters using 'camera_calibration' ROS package. The details involved are explained in the Readme.md file.

After the camera calibration, the mapping of the camera to the real world coordinates is done by multiplying the pixel information with the scaling factor. We are not using the z position (height) obtained from the detected arucos. Also, the angle obtained is limited to 0 to pi/2. We are considering (729,435) as the origin of the camera in the pixel coordinates.



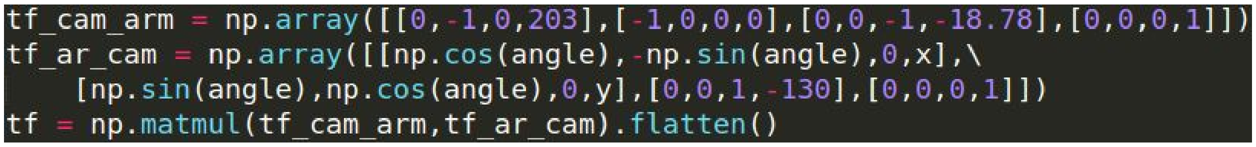
```python
x = (x - 729)*0.6
y = (y - 435)*0.5
angle = angle*np.pi/180

if(angle>np.pi/2):
    angle = angle-np.pi/2
elif(angle<-np.pi/2):
    angle = angle+np.pi
elif(angle>=-np.pi/2 and angle<-0):
    angle = angle+np.pi/2
```

Fig. 5

The camera coordinates were converted into the real world coordinates in the Lynx base frame using the transformation matrices. The constants set are obtained by actual measurements.

$$T_{aruco}^{arm} = T_{cam}^{arm} \times T_{aruco}^{cam}$$



```python
tf_cam_arm = np.array([[0,-1,0,203],[-1,0,0,0],[0,0,-1,-18.78],[0,0,0,1]])
tf_ar_cam = np.array([[np.cos(angle),-np.sin(angle),0,x],\
    [np.sin(angle),np.cos(angle),0,y],[0,0,1,-130],[0,0,0,1]])
tf = np.matmul(tf_cam_arm,tf_ar_cam).flatten()
```

Fig. 6

Planning

We are planning in configuration space using RRT. For detecting collisions, we are modelling the robot as a series of rectangular prisms of measured dimensions. Then we discretize the prism faces into lines of known spacing. Then we check each line with obstacles for collision using detectCollision.p. For self collisions, we use the same function taking the non-adjacent link prisms as obstacles.

Simplifying Assumptions:

1. The spacing to discretize the prism face is fine enough to detect a collision. The obstacles space had a minimum dimension of 80mm, while our spacing is 3mm, which is reasonable.
2. The robot links are modelled as rectangular prisms for simplicity which leads to a conservative estimate. This simplification helps while checking for self-collision using detectCollision.p
3. Joints variables of joints 4, 5 and 6 are kept at zero, as they do not affect the position of the wrist centre.

Pseudocode

```
1. Define Goal and Start in C-space
2. Append them to T_goal and T_start respectively, defined as child arrays
3. Initiate parent arrays which store the index of the closest node in the child array
4. Sample a random point q_i in C-space
5. If NOT ( q_i is in O-space)
        a. Compute q_closest for which distance between q_i and T_start is minimum
        b. Join q_closest and q_i with a line
        c. Discretize line into 10 points
        d. If (each point on the line is not in O-space)
                i. Append q_i to T_start
                ii. Store the index of q_closest in parent array
6. Repeat Step 5 for T_goal
7. If q_i is connected to both T-start and T_goal
        a. Terminate
        b. Stitch the path from q_i to Start, and q_i to Goal
        c. Return the path
8. Else repeat from step 4
```

Pseudocode for collision check with obstacles for a configuration q:

```
1. Define modelled link prism corners in joint frames
2. Discretize each face of the prism into equally spaced lines.
3. Get the endpoints of the lines in global frame running Forward Kinematics
4. Check collision between these lines and the obstacles in the environment
5. Return 1 if a collision happens, else return 0
```

Pseudocode for self-collision check for a configuration q:

```
1. Consider one of the link prisms as an obstacle
2. Transform the discretized lines of other prisms into the local frame of the obstacle link prism
3. Check collision between these lines and obstacle link prism
4. Return 1 if a collision happens, else return 0
```

We do this only for non-adjacent links, as adjacent links never collide within joint limits

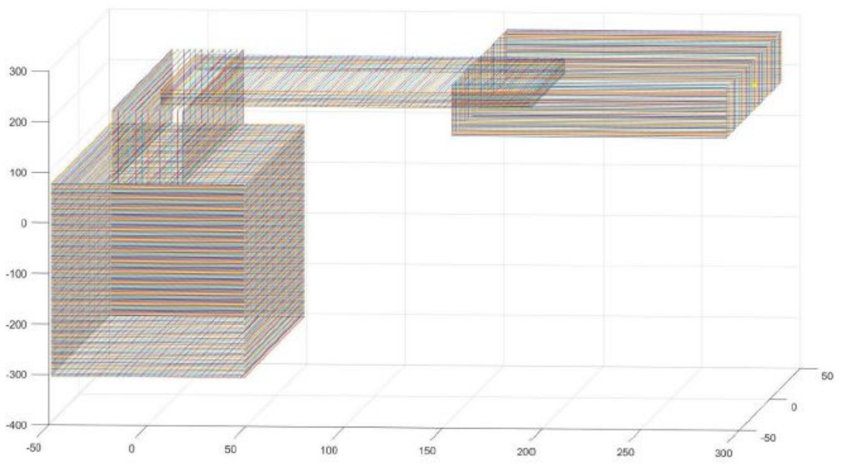


Fig. 7 Discretized Robot Model

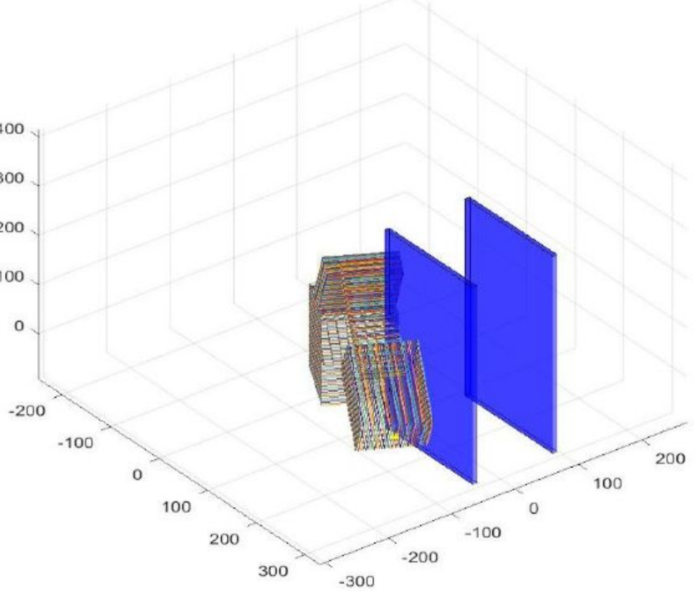


Fig. 8 Robot colliding with an obstacle

The inputs to the planner are the transformation matrices (first three rows) of the start and goal locations. The ROS node in MATLAB subscribes to the topics '/start_tf' and '/goal_tf' which contains the transformation matrices of all the cubes in the arena. The planner then gets the joint variables of these points using Inverse Kinematics. We use only Inverse Position Kinematics to get the wrist centre (i.e. the first three joints) to a fixed height (80 mm) above the cube. The planner then charts a path from the current location to the point above the cube. The last three joints are kept static in this process.

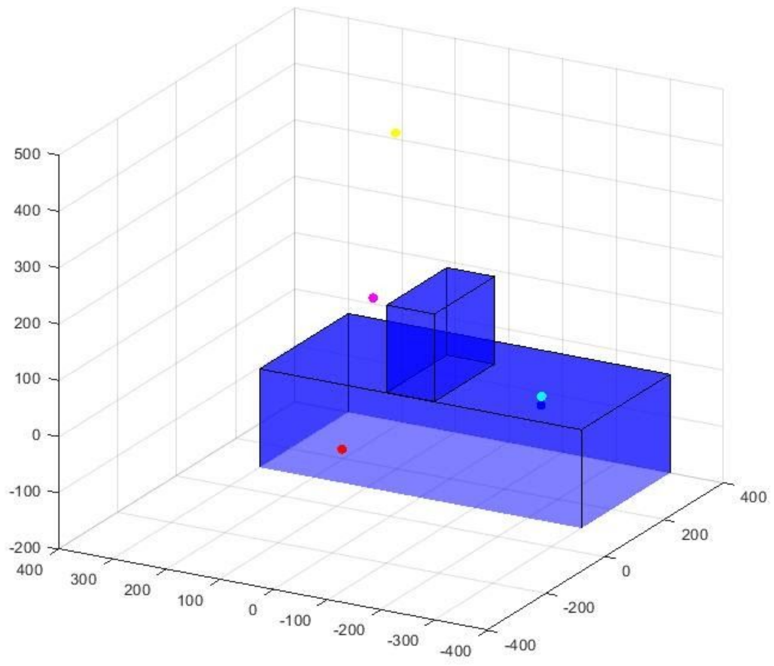


Fig. 9 Scatter plot of the end effector points
Magenta = start, Cyan = end, Yellow = chosen waypoints, Red= self collision,
Blue = collision with obstacles

After reaching the required configuration, the last three joints variables are varied such that the gripper is vertical (joint 4), the orientation of the gripper aligns with the faces of the cube (joint 5), and the gripper is opened (joint 6). The wrist centre then moves down a fixed height, the cube is picked up and reverts back to a point 80 mm above the cube's start position.

The RRT planner is then used again to move the robot to the goal position and the last three joints are again changed in a similar manner to place the block in the desired position. These tasks are repeated until all the cubes are placed in the required positions.

Execution

The planning module is implemented in MATLAB. These are the files we had written for LAB-2. In order to avoid rewriting of these files, we integrated ROS with MATLAB. ROS master with the sensing and execution runs on one system while the planner on MATLAB runs on another system. Both the systems are connected to the common wifi network. The connection can be established by using the export ROS IP command in every terminal of the system running the roscore. The bird view of the system is shown in Fig. 10

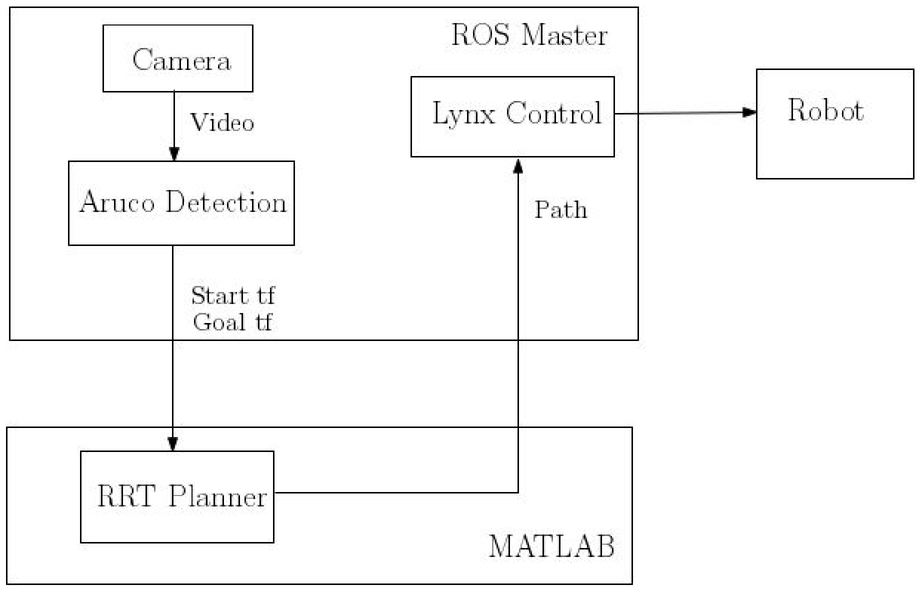


Fig.10 System Setup

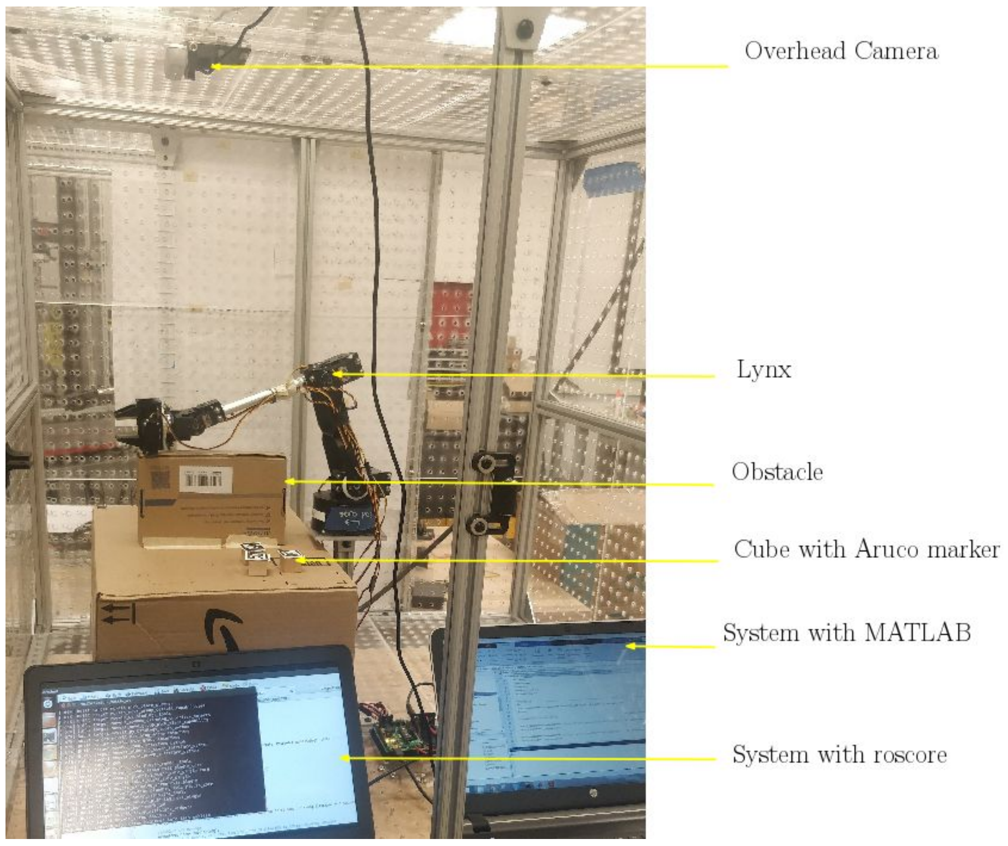


Fig. 11 Hardware Setup

‘lynx_control’ is the ROS package written for sending data to the Lynx robot. ‘lynxDriver.py’ is the driver script that communicates to the Lynx microcontroller via the serial communication. ‘pathSender’ subscribes to the topic ‘/path’ published by the MATLAB ROS node and publishes the joint parameters on the topic ‘/joint_variables’.

Fig. 12 is the rqt_graph of the entire system. It describes the various nodes communicating with each other on different topics with suitable message types.

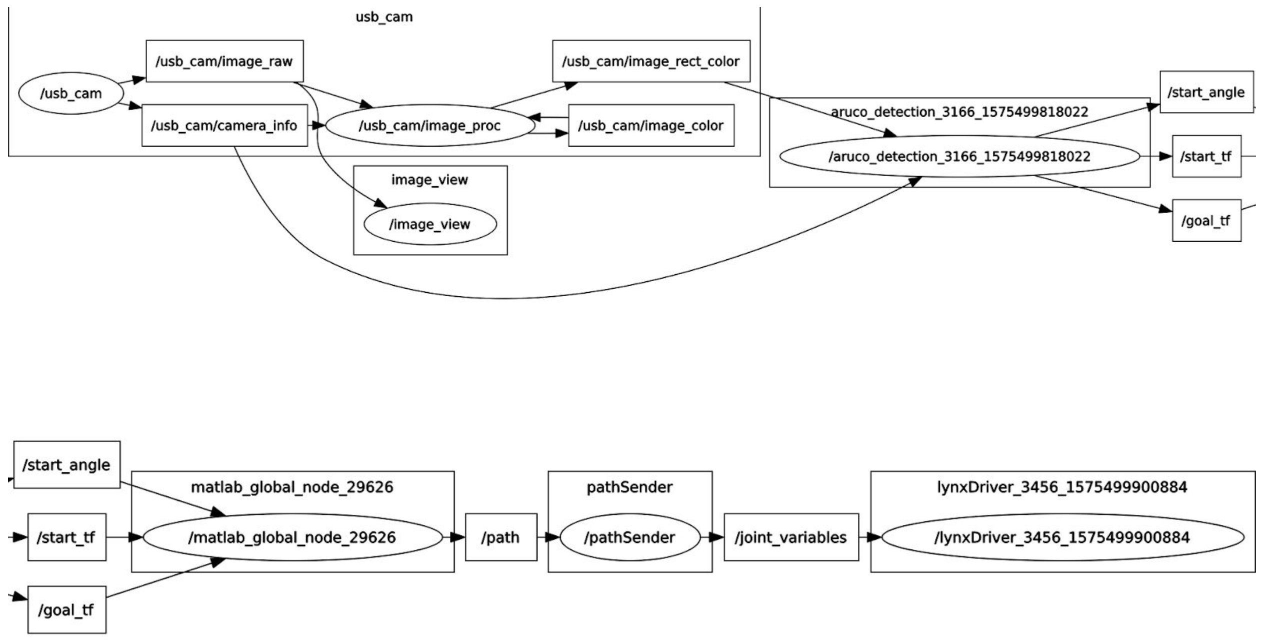


Fig. 12 rqt_graph

We performed experiments with three cubes and a single obstacle. The link to this video can be found here: Link

Conclusions

The ROS-MATLAB integration framework was set up in this work. Although the experiments were performed with only three cubes it is easily scalable.

Calibration of the camera was one of the major factors which controlled the success rate of performing the task. Although the camera localised the cubes reasonably well in the regions around its principal axis, as the cube was moved towards the extremes of the camera field of view, the values obtained were not linear in nature. Also, the error due to the linear approximation was not consistent in x and y axes of the camera. Initially, we decided to place the cubes at different heights, but the aruco marker was not sensitive to the height changes. An error of about 10cm was present in z-axis which made it unsuitable to localize the cubes at different heights. Along with this, the x and y coordinates of the cubes from the camera changed at different heights even though they did not change in the real world coordinates. These problems made us discard the idea of placing the cubes at different heights but instead, we introduced static obstacles in the environment whose locations were known. So now the planning problem became slightly more complex in nature and we used the RRT to handle the task.

Another major factor was the servo parameters like the scales and offsets which affected the performance of our system. These parameters were not linear in nature which prevented the Lynx from reaching the points as specified in the transformation matrix. Gravity effects came into the picture when the arm was extended out from the base frame. The motors could not provide enough torque to hold the arm upright at a given position which in turn induced a sag. As a result, we were unable to place the cubes in the regions of its workspace far away from the base frame. Thus the robot and camera parameters needed to be calibrated in a particular region for the robot to reach the required position.

Integrating ROS and MATLAB was another challenge. Initially, we had written our custom data structures to publish the data on the topics. But using these custom messages in MATLAB was not a straightforward approach hence we had to convert them to standard types supported by MATLAB. Also, the commands and syntax to write ROS nodes were not similar in MATLAB.

Low-level hardware scripts or the driver scripts for the Lynx had to be written to control the robot. This required us to understand the protocol used to communicate with the Lynx microcontroller board. In short, we had to convert the lynxServo.m file and the corresponding files to the ROS compatible format.

Initial days of our work were spent on understanding the working of ROS and scripting custom files followed by figuring out how to write the driver scripts for Lynx. During the mid-time, we had to understand the computer vision part and also integrate ROS and MATLAB. The later part of the project involved calibration and experimentation which consumed a major chunk of the time.

To sum up, at the level of logic implementation and setting up the framework, we consider that our work is complete. If provided with a better calibrated system, the success rate of our project would depend solely on the success rate of the planner. This opens up the space of exploring robust methods to calibrate the camera and the servos which can model the non-linearity in the system.

Appendix

ROS scripts:

1) aruco.py - Node that detects the aruco markers and publishes the aruco information
2) pathSender.cpp - Node that subscribes to the path and sends a set of joint values to the Lynx driver
3) lynxDriver.py - Node that connects to the Lynx board via serial communication.
4) aruco_cal.py - Node that can be used to calibrate the camera origin and scaling parameters

MATLAB scripts and functions :

1) ROS_Node.m - Script that contains all the publishers and subscribers for communication between the two terminals. Calls the path planner and pick/place functions and sends the evaluated path to ROS
2) CalculateIK_sol.m - Script returns first three joint variables for a given transformation matrix
3) Grip.m - Appends configurations which pick up the cube to the path
4) Release.m - Appends configurations which place the cube to the path
5) Rrt.m - Takes start and goal configurations as input and returns a path between them using RRT
6) Linesgen.m - Script that models the robot links into rectangular prisms with equally spaced lines in robot frame
7) isRobotCollided.m - Script that detects if a configuration of the Lynx is in collision with any obstacles on the map
8) isRobotSelfCollided.m - Script that detects if a configuration of the Lynx is in self-collision
9) Local_to_global.m - Creates model of robot in global frame in a given configuration
10) Randgen - Spawns a configuration within joint limits
11) Stitch - Stitches chosen configuration into a single path from start to goal

The detailed description of how to run the system is explained in the readme.md file in the codes folder.

References

1) wiki.ros.org
2) stackoverflow.com
3) Aruco Marker Documentation
4) Camera Calibration
5) MATLAB Documentation