Final Report: Introduction to Robotics Lab

Syeda Emaan*, Fatima Bint Ejaz†,Mannan Muhammad Rangoonia‡
*Habib University, Pakistan
Email: se08148@st.habib.edu.pk
†Habib University, Pakistan
Email: fa08519@st.habib.edu.pk
‡Habib University, Pakistan
Email: mr07882@st.habib.edu.pk

***Abstract*—This research paper presents an integrated robotic system for automated cube recognition and manipulation using a PhantomX Pincher robot arm, Intel RealSense depth camera, and MATLAB programming environment. The system combines computer vision techniques for object detection and color classification with forward and inverse kinematics solutions for precise manipulation of colored cubes in a defined workspace. This paper details the development of a comprehensive robotic framework including workspace analysis, mathematical modeling, vision-based object detection, and motion planning. Experiment results demonstrate the system's ability to successfully detect, identify, pick, and place cubes with high accuracy within the robot's reachable workspace. This work represents a foundation for broader applications in robotic manipulation and human-robot interaction systems**

***Index Terms*—Phantom X Pincher, Robot Manipulator, Pick and Place**

I. Introduction

Robotic manipulation in unstructured environments represents one of the fundamental challenges in robotics research. The ability for a robot to perceive, identify, and interact with objects in its environment has applications across numerous domains, including manufacturing, healthcare, domestic assistance, and education. This project aims to develop an integrated system that combines computer vision and robotic manipulation to autonomously detect, pick, and place colored cubes within a defined workspace. The core objective of this research is to create a robust system that can:

- Detect colored cubes within the robot's workspace using an RGB-D camera
- Determine the precise spatial coordinates of these cubes
- Plan and execute appropriate motion sequences to pick up and relocate these objects
- Complete these tasks with minimal human intervention

The significance of this project extends beyond the specific task of cube manipulation. The underlying techniques developed here—combining 3D perception, spatial reasoning, and physical manipulation—form the foundation for more complex robotic applications in real-world environments. For instance, similar approaches can be employed in automated assembly, sorting operations, assistive robotics, and educational platforms for robotic learning.

This paper is structured as follows: Section 2 describes the methodology and system design, detailing the kinematic modeling, computer vision pipeline, and motion planning algorithms implemented. Section 3 presents the experimental results, quantifying the performance and limitations of the system. Section 4 discusses conclusions drawn from this work and suggests directions for future research. Additional technical details and complete code implementations are provided in the appendices.

II. Design Details

A. System Architecture

The complete robotic system integrates hardware components and software modules to create a functional cube manipulation platform. Fig. 1 presents the functional architecture of the system.

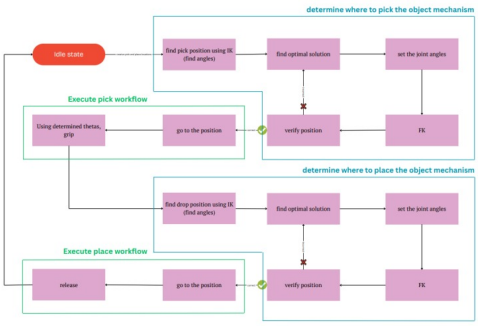

Fig. 1. FSM of the project pipeline

B. Overview

The complete robotic system integrates hardware components and software modules to create a functional cube manipulation platform.

The system consists of the following primary components:

Hardware Components:

- PhantomX Pincher robot arm with 5 servo motors
- Intel RealSense depth camera
- Colored cubes as manipulation targets
- Computing platform running MATLAB

Software Components:

- Perception module: Cube detection and localization
- Kinematics module: Forward and inverse kinematics solutions

- Motion planning module: Trajectory generation and execution
- Control module: Interface with hardware components

The system operates through a workflow that begins with environment perception, proceeds through object identification and localization, followed by motion planning and execution. The following sections give details for each of these components.

C. Kinematics Modelling

1) Frame Assignment: We assigned base frame, intermediate frames, and end effector frames to the PhantomX Pincher arm as shown in Fig. 2.

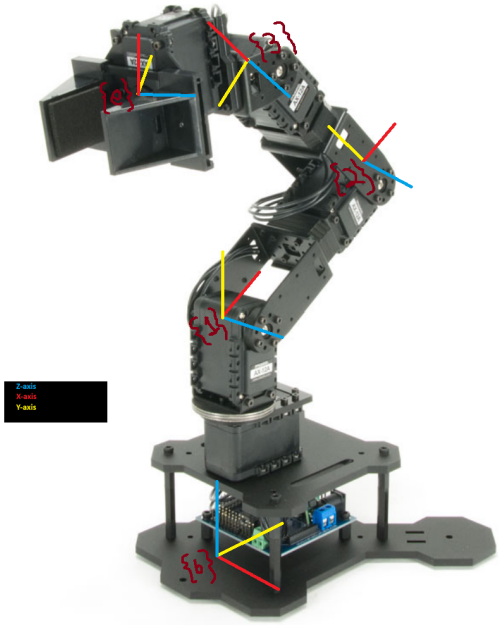

Fig. 2. POE frame assignment

2) Forward Kinematics: Forward kinematics involves determining the end-effector position and orientation given the joint angles of the robot. For the PhantomX Pincher robot, we established a Denavit-Hartenberg (DH) parameter framework. The parameters are summarized in Table I.

TABLE I
DH PARAMETERS FOR PHANTOMX PINCHER ROBOT

Joint	a_i (cm)	α_i (rad)	d_i (cm)	Q_i
1	0	$\pi/2$	14.8	Q_1
2	10.5	0	0	Q_2
3	10.5	0	0	Q_3
4	7.8	0	0	Q_4

Using these parameters, we compute the transformation matrices between frames using the standard DH transformation:

$$T_i^{i-1} = \begin{bmatrix} \cos Q_i & -\sin Q_i \cos \alpha_i & \sin Q_i \sin \alpha_i & a_i \cos Q_i \\ \sin Q_i & \cos Q_i \cos \alpha_i & -\cos Q_i \sin \alpha_i & a_i \sin Q_i \\ 0 & \sin \alpha_i & \cos \alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The full transformation from base to end-effector:

$$T_0^4 = T_0^1 \cdot T_1^2 \cdot T_2^3 \cdot T_3^4$$

Pseudocode:

```
function FK(Q1, Q2, Q3, Q4):
    Define a, alpha, d using DH table
    Compute T01, T12, T23, T34
    using DH transformation
    Multiply: T04 = T01 * T12 * T23 * T34
    Extract x, y, z, R from T04
    Return x, y, z, R
EndFunction
```

To visualize the robot's workspace, we sampled the joint space and computed the corresponding end-effector positions using the forward kinematics.The plot shown in Fig. 3 is the desired workspace, however, it is not the reachable workspace in our case as the robot's base is situated on a desk. In our case, the reachable workspace would be the upper hemisphere of the workspace.

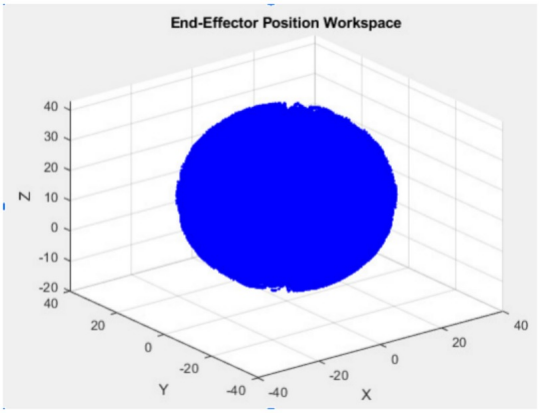

Fig. 3. End Effector Workspace

3) Inverse Kinematics: Inverse kinematics involves determining the joint angles required to position the end-effector at a desired location with a specific orientation. For the PhantomX Pincher robot, we developed a geometric approach to solve this problem. Given a desired end-effector position (x, y, z) and orientation angle phi, we derived the following solution:

3.1: First, we computed the wrist center by offsetting from the desired end-effector position based on the orientation, which is evident from Fig. 4:

$$r = \sqrt{x^2 + y^2}$$

$$s = z - d_1$$

$$u = r - a_4 \cos(\phi)$$

$$v = s - a_4 \sin(\phi)$$

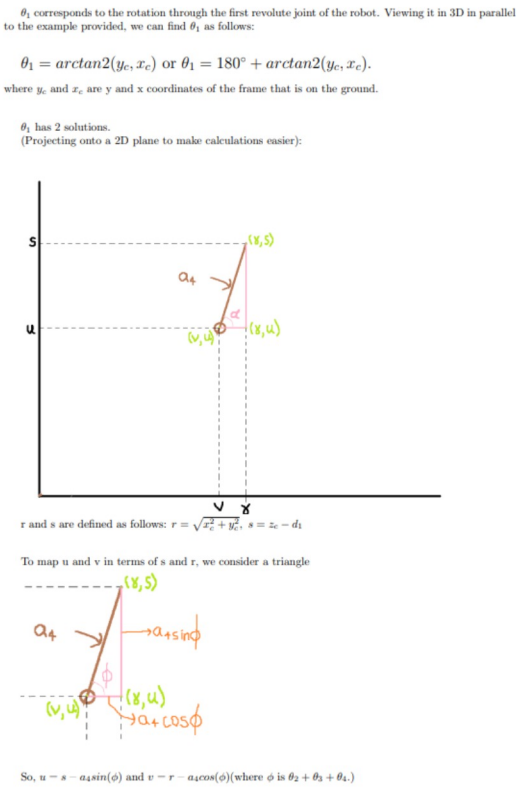

Fig. 4. Wrist center offset computation

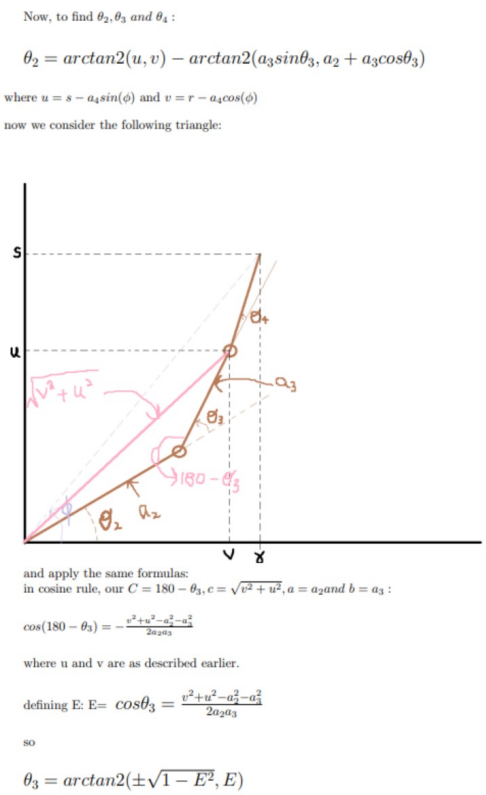

Fig. 5. Calculation of All Joint Angles θ_1

3.2: For the first joint angle ($theta_1$), we used the arctangent of the end-effector's x-y projection:

$$\theta_{1_1} = \text{atan2}(y, x) \quad \theta_{1_2} = \pi + \theta_{1_1}$$

3.3: For the third joint angle ($theta_3$), we applied the law of cosines to the triangle formed by links 2 and 3:

$$\cos(\theta_3) = \frac{u^2 + v^2 - a_2^2 - a_3^2}{2a_2a_3}$$

$$\theta_{3_1} = \text{atan2}\left(\sqrt{1 - \cos^2(\theta_3)}, \cos(\theta_3)\right)$$

$$\theta_{3_2} = \text{atan2}\left(-\sqrt{1 - \cos^2(\theta_3)}, \cos(\theta_3)\right)$$

3.4: For the second joint angle ($theta_2$), we used the geometry of the arm configuration:

$$\theta_{2_1} = \text{atan2}(v, u) - \text{atan2}(a_3 \sin(\theta_{3_1}), a_2 + a_3 \cos(\theta_{3_1}))$$

$$\theta_{2_2} = \text{atan2}(v, u) - \text{atan2}(a_3 \sin(\theta_{3_2}), a_2 + a_3 \cos(\theta_{3_2}))$$

Finally, for the fourth joint angle ($theta_4$), we used the desired orientation:

$$\theta_{4_1} = \phi - \theta_{2_1} - \theta_{3_1} \qquad \theta_{4_2} = \phi - \theta_{2_2} - \theta_{3_2}$$

These equations provide multiple solutions for the inverse kinematics problem, as is typical for robotic manipulators. The complete set of solutions was organized into a $4 \ times 4$ matrix, with each row representing a distinct solution.

```
Function FindJointAngles(x, y, z, phi):
    Compute wrist center (u, v)
    Compute Q1 = atan2(y, x)
    Use cosine law for Q3
    Compute Q2 using arctangent decomposition
    Q4 =  - Q2 - Q3
    Return up to 4 valid configurations
EndFunction
```

The inverse kinematics solutions were verified by substituting them back into the forward kinematics equations and confirming that the resulting end-effector position matched the original input. Moreover, the outputs from inverse kinematics were also physically verified using a ruler. It was observed that solutions 1, 2, and 4 were consistently accurate, while solutions 2 and 3 occasionally yielded incorrect results—particularly due to errors in the calculation of Q_{3_2}.

4) Finding an Optimal Solution: Once multiple inverse kinematics solutions were computed, we needed to select the optimal one for execution. Our approach considered joint limits, the current configuration of the robot, and minimization of total joint movement.

The function `FindOptimalSolution` executes the following steps:

- **Generate Candidate Solutions:** All mathematically fea-

sible joint configurations are first generated by the inverse kinematics function. These include redundant or mirrored solutions.

- **Filter Invalid Configurations:** Each solution is checked using the `CheckJointLimits` function to ensure that the joint angles lie within the physical limits of the PhantomX Pincher (i.e., between $-150°$ and $150°$). Any configuration that violates these constraints is discarded.
- **Handle No-Solution Cases:** If no valid solution remains after filtering, the function reports failure and returns an empty output, ensuring that the robot does not execute unsafe commands.
- **Assess Motion Efficiency:** The current joint angles of the robot are obtained via `GetCurrentAngles`. For each valid solution, the absolute difference from the current configuration is computed. The total error across all joints is used as a cost metric.
- **Choose Optimal Path:** The solution with the minimum total joint deviation is selected. This minimizes mechanical effort, reduces unnecessary movement, and results in smoother, more efficient operation.

```
function FindOptimalSolution(x, y, z, )

    % Step 1: Compute all possible inverse
    kinematics solutions
    solutions ← FindJointAngles(x, y, z, )

    % Step 2: Validate each solution against
    robot's joint limits
    valid_solutions ← []
    for each sol in solutions:
        if CheckJointLimits(sol):
            valid_solutions.append(sol)

    % Step 3: If no valid solution exists,
    return empty and alert
    if valid_solutions is empty:
        print "No solutions found"
        return []

    % Step 4: Get the current joint
    configuration of the robot
    current_angles ← GetCurrentAngles()

    % Step 5: Compute total error from
    each valid solution to current
    joint angles min_error ←
    best_solution ← null

    for each sol in valid_solutions:
        error ← sum of absolute differences
        between sol and current_angles
        if error < min_error:
            min_error ← error
            best_solution ← sol

    % Step 6: Return the configuration with
    minimum total error
    return best_solution

EndFunction
```

D. Perception Pipeline

The perception pipeline is responsible for detecting colored cubes in the workspace and determining their positions in 3D space in the base frame of the robot arm.

1) Camera Coordinates: This pipeline consists of several stages, all handled by the function `GetLocationsOfCubes`:

```
function GetLocationsOfCubes()

    Initialize RealSense pipeline
    Capture color and depth frames
    Align frames
    Apply spatial and temporal filters
    Binarize color image → extract connected
                           components
    Filter based on size (500{800 px)

    For each object:
        Compute mean RGB → classify color
        Compute depth histogram → get Z value
        Extract centroid in (u, v) pixel space

    Return [u, v, z] for each object

EndFunction
```

Below, each stage of the pipeline is explained in detail.

1.1: Image Acquisition and Processing

First, we configure and initialize the Intel RealSense camera to capture both color (RGB) and depth information. The camera is set up with appropriate parameters for auto-exposure and white balance to ensure consistent color recognition. After allowing a few frames to stabilize, the function captures a synchronized frame set. These frames are aligned (color to depth) to ensure pixel correspondence.

To improve depth data quality, the function applies two filters:

- A spatial filter to smooth and fill holes in the depth map.
- A temporal filter to reduce temporal noise between frames.

After filtering, the color and depth data are extracted and reshaped into standard image formats. The depth values are scaled to real-world meters using the device's `depth_scale` parameter and later converted to centimeters for display and analysis.

1.2: Cube Detection and Color Classification

Using the RGB image, the function applies `imbinarize` to separate the objects from the background. It inverts the binary image and looks for connected components (`bwconncomp`) that correspond to individual blobs. These blobs are filtered based on pixel count (size thresholding between 500 and 800 pixels) to eliminate noise or irrelevant regions. A second connected component analysis is done on the refined binary image to isolate valid cuboid candidates.

The function then computes the mean red, green, and blue intensities of each component to classify them based on color—identifying red, green, or blue cubes. For each remaining object, the function examines its depth values and computes a histogram of non-zero values. It selects a threshold based on the dominant bin to represent the object's Z-depth. Only pixels below this depth are considered valid to form a clean mask of the object, which is then used to refine detection.

1.3: Centroid Calculation

The centroid of each object is computed by averaging the row and column indices of the pixels within each component. These centroid positions in pixel space, along with the previously calculated Z-depth values (scaled to mm), form the camera-space coordinates $[x, y, z]$. These are saved as a list of coordinate triplets.

2) Camera to World Coordinate Transformation: Finally, the detected cube positions are transformed from the camera coordinate system to world coordinates aligned with the robot's base frame, using the camera's intrinsic parameters. In this setup, the negative X-axis of the camera frame corresponds to the positive X-axis of the robot's base frame. Therefore, a simple inversion of the X-coordinate from the camera data is sufficient to account for this rotational alignment.

```
function CameraToWorld(Coordinates):
    Retrieve camera intrinsics
    (fx, fy, cx, cy)
    For each coordinate [u, v, Z]:
        X = -(u - cx) * Z / fx
        Y = (v - cy) * Z / fy
        Z = 0  % flat plane assumption
    Return [X, Y, Z] in cm
EndFunction
```

This transformation allows us to express the cube positions in the robot's operational space, making it possible to plan and execute manipulation tasks.

E. Complete Pick and Place Pipeline

This motion planning and execution module implements the finite state machine (FSM) shown in Figure 1 to coordinate the robotic arm's movements for picking and placing cubes.

The FSM has several states that guide the robot through the task sequence:

- **State 0 – Home Position Initialization:** The robot moves to a safe home configuration [0, 0, $-\pi/4$, $-\pi/2$, 0], ensuring a known starting pose.
- **State 1 – Move Above Pick Position:** The robot increases the Z-coordinate of the pick point by 16 units to approach safely from above, computing the optimal joint angles.
- **State 2 – Go to Above Pick Position:** The robot moves to this configuration using `setPosition`. A short pause allows verification via `arb.getpos()`.
- **State 3 – Move Down to Pick:** The robot lowers the end effector by 13 units to approach the object closely, using updated IK solutions.
- **State 4 – Grasp the Object:** The robot uses `PositionJaw` and `setPosition` to close the gripper. It checks the servo position to confirm a successful grasp.
- **State 5 – Move Above Place Position:** The Z-value is again increased by 16 units to safely lift the object and transition to the next state.
- **State 6 – Go to Above Place Position:** The robot moves above the target drop location while holding the object.
- **State 7 – Lower to Place Position:** The robot decreases the Z-coordinate by 13 units to prepare for releasing the object.
- **State 8 – Release the Object:** The gripper opens slightly (value 0.8) to let go of the object. No confirmation is added, but this could be improved with visual feedback.
- **State 9 – Retract Upward After Placement:** Finally, the robot moves upward once more by increasing the Z-coordinate by 12 units, retreating from the object to a safe height. This ends the pick-and-place task and prepares the robot for a potential next command. The gripper remains slightly open, indicating it is ready for the next pick operation.

```
function PickAndPlace(PickCo, PlaceCo):

    State ← 0

    while task not complete:

        switch State:

            case 0: Move to home

            case 1: Compute pre-grasp pose

            case 2: Hover over cube

            case 3: Lower to cube

            case 4: Grasp cube
```

```
            case 5: Compute pre-place pose

            case 6: Hover over place location

            case 7: Lower to place

            case 8: Release cube

            case 9: Retreat

EndFunction
```

III. Results

A. Performance Metrics

We conducted two trials to assess the performance of our system. The first trial focused on the boundary of the robot's workspace, while the second included cubes both at the boundary and inside the workspace.

Trial 1: Boundary of Workspace: In this trial, we placed 3 blue cubes, 3 green cubes, and 1 red cube at the boundary of the robot's workspace. The results are summarized in Table II.

TABLE II
Results of Trial 1

Metric	Performance
Perception Accuracy	100% (All 7 cubes detected with correct color identification)
Position Accuracy	Accurately determined real-world positions, verified by physical measurements
Picking Success Rate	90% of cubes picked accurately; some inverse kinematics solutions had an offset in X and Y coordinates
Picking Offset	±0.5 cm in X and Y for some cases
Placing Accuracy	100% (All cubes placed at the correct position)

Link to Video Run: https://youtube.com/shorts/6sCii5KhPtw?feature=share

Trial 2: Boundary and Inside Workspace: In the second trial, we placed 3 blue cubes, 2 green cubes, and 2 red cubes both on the boundary and inside the robot's workspace. The results are summarized in Table III.

TABLE III
Results of Trial 2

Metric	Performance
Perception Accuracy	100% (All 7 cubes detected with correct color identification)
Position Accuracy	Accurately determined real-world positions, verified by physical measurements
Picking Success Rate	90% success for reachable cubes; remaining 10% had slight X-Y offsets
Workspace Limitation	Unable to reach cubes in negative Y-axis region of base frame

Link to Video Run: https://youtube.com/shorts/mL192KKHpy0?feature=share

B. Analysis of Results

1) Quantitative Analysis: The system demonstrated high accuracy in both perception and manipulation tasks:

- **Perception Performance:**
  - 100% detection rate for all cubes within the camera's field of view
  - Accurate color classification for all detected cubes
- **Manipulation Performance:**
  - 90–100% success rate for picking cubes within the reachable workspace
  - 100% success rate for placing cubes at designated positions
  - Average positional error of ±0.5 cm during picking
- **Execution Time:**
  - Average of 40–50 seconds per complete pick-and-place operation
  - Perception pipeline processing time of 3–5 seconds per frame

2) Qualitative Analysis: Beyond the quantitative metrics, several qualitative insights were observed:

- **Workspace Limitations:**
  - Difficulty reaching positions in the negative Y-axis region
  - Likely caused by joint 2 colliding with the base, due to invalid IK solutions
- **Offset Sources:**
  - Camera calibration imprecision
  - Depth measurement errors from RealSense camera
  - Mechanical backlash in joints
  - Small IK inaccuracies
- **System Robustness:**
  - Consistent performance across trials with varying cube arrangements
  - Robust color classification under different lighting (auto-exposure and white balance)
  - Reliable task execution due to the finite state machine architecture
- **Tradeoffs:**
  - Slower arm movement improved accuracy but increased total task time

IV. Conclusion and Future Work

This project successfully implemented an integrated robotic system capable of autonomously detecting, picking, and placing colored cubes using a PhantomX Pincher robot arm and an Intel RealSense camera. The system combined computer vision techniques for object detection and localization with kinematic modeling for precise manipulation.

Our approach demonstrated several key achievements:

- Development of accurate forward and inverse kinematics solutions for the PhantomX Pincher arm
- Implementation of a robust perception pipeline for cube detection and localization

- Design of an effective finite state machine for coordinating pick-and-place operations
- Achievement of high success rates in both object detection and manipulation tasks
- Reliable system performance within the robot's reachable workspace, with minor limitations in reaching certain objects

V. Digital Material

https://github.com/mr07882/RoboticsProject.git

References

[1] B. Memon, "EE366L/CE366L: Introduction to Robotics Lab Handbook," Jan. 2024. [Online].

[2] M. W. Spong, S. Hutchinson, and M. Vidyasagar, *Robot Modeling and Control*, 2nd ed. Wiley, 2020.

Appendix

The Matlab code for the entire project is below:

```
function Coordinates =
GetLocationsOfCubes()
Pipe = realsense.pipeline();
Colorizer = realsense.colorizer();
CFG = realsense.config();
streamType = realsense.stream('depth');
formatType = realsense.format('Distance');
CFG.enable_stream(streamType, formatType);
streamType = realsense.stream('color');
formatType = realsense.format('rgb8');
CFG.enable_stream(streamType, formatType);
Profile = Pipe.start();
Device = Profile.get_device();
Name =
Device.get_info(realsense.camera_info.name);
DepthSensor =
Device.first('depth_sensor');
RGBSensor =
Device.first('roi_sensor');
DepthScale =
DepthSensor.get_depth_scale();
OptionType =
realsense.option('visual_preset');
DepthSensor.set_option(OptionType, 9);
OptionType =
realsense.option('enable_auto_exposure');
RGBSensor.set_option(OptionType, 1);
OptionType =
realsense.option('enable_auto_white_balance');
RGBSensor.set_option(OptionType, 1);
for i = 1:5
Frames = Pipe.wait_for_frames();
end
AllignToDepth = realsense.
align(realsense.stream.depth);
Frames = AllignToDepth.process(Frames);
Pipe.stop();
Depth = Frames.get_depth_frame();
Width = Depth.get_width();
Height = Depth.get_height();
Spatial = realsense.spatial_filter(.5, 20, 2, 0);
Depth_p = Spatial.process(Depth);
TemporalFilter =
realsense.temporal_filter(.13, 20, 3);
Depth_p = TemporalFilter.process(Depth_p);
Color = Frames.get_color_frame();
DepthColor = Colorizer.colorize(Depth_p);
Data = DepthColor.get_data();
Image = permute(reshape(Data',
[3, DepthColor.get_width(),
DepthColor.get_height()]), [3 2 1]);
Data3 = DepthScale * single(Depth_p.get_data());
ig = permute(reshape(Data3',
[Width, Height]), [2 1]);
Data2 = Color.get_data();
Im = permute(reshape(Data2', [
3, Color.get_width(),
Color.get_height()]), [3 2 1]);
BW = imbinarize(Im);
figure
imshowpair(Im, BW, 'montage')
title(" Imbinarize method ")
TempImage = ~(BW(:,:,1)
& BW(:,:,2) & BW(:,:,3));
cc4 = bwconncomp(TempImage, 4);
L4 = labelmatrix(cc4());
RGBLabel = label2rgb(L4, @copper, "c", "shuffle");
Size = size(cc4.PixelIdxList{1});
for k = 1:cc4.NumObjects
Size = size(cc4.PixelIdxList{k});
if Size(1) < 500 || Size(1) > 800
TempImage(cc4.PixelIdxList{k}) = 0;
end
end

figure;

imshow(TempImage)

title('NO CUBES DETECTED BUDDY !')

cc4_ = bwconncomp(TempImage, 4);

Red = zeros(1, cc4_.NumObjects);
Green = zeros(1, cc4_.NumObjects);

Blue = zeros(1, cc4_.NumObjects);

RedImage = Im(:,:,1);

GreenImage = Im(:,:,2);

BlueImage = Im(:,:,3);
```

```
for k = 1:cc4_.NumObjects

MeanRedPixel =
mean(RedImage(cc4_.PixelIdxList{k}));

MeanGreenPixel =
mean(GreenImage(cc4_.PixelIdxList{k}));

MeanBluePixel =
mean(BlueImage(cc4_.PixelIdxList{k}));

if MeanRedPixel > 65

Red(k) = 1;

end

if MeanGreenPixel > 65

Green(k) = 1;

end

if MeanBluePixel > 67

Blue(k) = 1;

end

end



NewIg = ig * 100;

BlankImage = TempImage;

BlankImage(:,:) = 0;

ZValue = zeros(1, cc4_.NumObjects);

for k = 1:cc4_.NumObjects

NonZeroPixel = [];

for j = 1:length(cc4_.PixelIdxList{k})

if NewIg(cc4_.PixelIdxList{k}(j)) ~= 0

NonZeroPixel(end+1) =
NewIg(cc4_.PixelIdxList{k}(j));

end

end

non_zero_his = histogram(NonZeroPixel, 5);

[MaxPixels, idx] = max(non_zero_his.Values);

if idx + 2 <= length(non_zero_his.BinEdges)

MaxThreshold = non_zero_his.BinEdges(idx+2);

else

MaxThreshold = non_zero_his.BinEdges(idx+1);

end

ZValue(k) = MaxThreshold;

for j = 1:length(cc4_.PixelIdxList{k})

if NewIg(cc4_.PixelIdxList{k}(j))
<= MaxThreshold

BlankImage(cc4_.PixelIdxList{k}(j)) = 1;

end

end

end

figure;

imshowpair(TempImage, BlankImage, 'montage');

cc4_for_cen = bwconncomp(BlankImage);

CentroidCoordinates =
zeros(2, cc4_for_cen.NumObjects);

for i = 1:cc4_for_cen.NumObjects

col_ =
mod(cc4_for_cen.PixelIdxList{i}, 480);

row_ =
ceil(cc4_for_cen.PixelIdxList{i} / 480);

row_fin = ceil(mean(row_));

col_fin = ceil(mean(col_));

CentroidCoordinates([1,2], i) =
[row_fin, col_fin];

BlankImage(row_fin, col_fin) = 0;
```

```
end

figure;

imshow(BlankImage)



save tmp1

Length = size(CentroidCoordinates);

for k = 1:Length(2)

CentroidCoordinates(1, k) =
CentroidCoordinates(1, k);

CentroidCoordinates(2, k) =
CentroidCoordinates(2, k);

end

Coordinates = [];

for k = 1:Length(2)

Coordinates{k} = [CentroidCoordinates
(1, k),
CentroidCoordinates(2, k),
ZValue(k)*10];

end

Coordinates = Coordinates';

end

function [WorldCoordinates] =
CameraToWorld(CamCoordinates)

Pipe = realsense.pipeline();

CFG = realsense.config();

CFG.enable_stream(realsense.stream.depth);

profile = Pipe.start(CFG);

depth_stream =
profile.get_stream(realsense.stream.
depth).as('video_stream_profile');

intrinsics =
depth_stream.get_intrinsics();

Pipe.stop();

% Extract intrinsic parameters

fx = intrinsics.fx;
% Focal length in x (pixels)

fy = intrinsics.fy;
% Focal length in y (pixels)

cx = intrinsics.ppx;
% Principal point x (pixels)

cy = intrinsics.ppy;
% Principal point y (pixels)



WorldCoordinates =
zeros(3, length(CamCoordinates));

for k = 1:length(CamCoordinates)

Z = CamCoordinates{k}(3);

world_X =
-(CamCoordinates{k}(1)-cx) * Z / fx;

world_Y =
(CamCoordinates{k}(2)-cy) * Z / fy;

WorldCoordinates(:, k) =
[world_X/10; world_Y/10; 0];

end

end

function IsValid = CheckJointLimits(Q)

JointLimits = [-150, 150];

IsValid = all(Q >= JointLimits(1)
& Q <= JointLimits(2));

end



function Angles = GetCurrentAngles()

arb =
Arbotix('port', 'COM4', 'nservos', 5);

Angles = arb.getpos();
```

```
end


%BASIC INVERSE KINEMATICS FROM PREVIOUS LAB


function AllSolutions =
FindJointAngles(x, y, z, phi)

a2 = 10.5;

a3 = 10.5;

d1 = 14.8;

a4=7.8;


%WRIST CENTER COORDINATES

r = sqrt(x^2 + y^2);

s = z - d1;


u=r-a4*(cos(phi));

v=s-a4*(sin(phi));


%THETA 1 SOLUTIONS:

Q1_1 = atan2(y, x);

Q1_2 = pi+Q1_1;


%COS Q3 USING COSINE RULE

cosQ3 =
(u^2 + v^2 - a2^2 - a3^2) /
(2 * a2 * a3);


%THETA 3 SOLUTIONS

Q3_1 =
atan2(real(sqrt(1 - cosQ3^2)), cosQ3);

Q3_2 =
atan2(-real(sqrt(1 - cosQ3^2)), cosQ3);


%THETA 2 SOLUTIONS

Q2_1 = atan2(v, u) -
atan2(a3 * sin(Q3_1), a2 + a3
* cos(Q3_1));

Q2_2 = atan2(v, u) -
atan2(a3 * sin(Q3_2), a2 + a3
* cos(Q3_2));


%%THETA 4 SOLUTIONS

Q4_1 = phi - Q2_1 - Q3_1;

Q4_2 = phi - Q2_2- Q3_2;


%4X4 SOLUTIONS MATRIX

AllSolutions =
[Q1_1+pi/2 pi/2-Q2_1 -Q3_1 -Q4_1;

Q1_1+pi/2 pi/2-Q2_2 -Q3_2 -Q4_2;

Q1_2+pi/2 pi/2-(pi-Q2_1) -Q3_2 Q4_2;

Q1_2+pi/2 pi/2-(pi-Q2_2) -Q3_1 Q4_2;];


%WRAPPING TO PI

AllSolutions =
mod(AllSolutions+pi,2*pi)-pi;


end


function MySolution =
FindOptimalSolution(x, y, z, phi)

%ALL INVERSE KINEMATICS SOLUTIONS
```

```
solutions =
FindJointAngles(x, y, z, phi);


if y < 0

MySolution = solutions(1,:);

else

MySolution = solutions(4,:);

end


if abs(MySolution(4)) > pi/2

MySolution(4) =
sign(MySolution(4)) * (pi/2);

end


%FILTERING OUT NON REALISABLE SOLUTIONS
WHICH ARE OUTSIDE THE JOINT LIMITS

validSolutions = cellfun
(@CheckJointLimits,
num2cell(solutions, 2),
'UniformOutput', false);

validSolutions =
cell2mat(validSolutions);


%EXTRACTING VALID SOLUTIONS ONLY

valid_solutions_mask =
all(validSolutions , 2);

valid_solutions =
solutions(valid_solutions_mask, :);


if isempty(valid_solutions)

solution = [];

disp("NO SOLUTIONS FOUND BUDDY. SORRY!");
return;

end


%GETTING CURRENT JOINT
ANGLES OF THE ROBOT

CurrentJointAngles = GetCurrentAngles();


%FINDING THE ABSOLUTE
ERRORS OF ALL THE SOLUTIONS

Delta = abs(solutions -
CurrentJointAngles(1, 1:4));

%TOTAL ERROR FOR EACH SOLUTION

TotalError = sum(Delta, 2);


%GET THE OPTIMAL SOLUTION
WITH MINIMUM ERROR

[~, idx] = min(TotalError);

solution = solutions(idx, :);


end


function Q5 =
PositionJaw(Position)

%LINK LENGTHS IN CM

L1 = 0.868;

L2 = 2.591;


MinX = abs(L1 - L2);

MaxX = (L1 + L2) ;


if Position < MinX ||
```

```
Position > MaxX

warning('Invalid jaw position.
Must be
between %.2f and %.2f mm.',
MinX, MaxX);

return;

end


x = Position;

cosQ2 = (x^2 - L1^2 - L2^2) /
(2 * L1 * L2);


sin_theta2 = -sqrt(1 - cosQ2^2);

k1 = L1 + L2 * cosQ2;

k2 = L2 * sin_theta2;

Q5 = atan2(0, x) - atan2(k2, k1);


end



function SetPosition(JointAngles,
GripperAngle)

% Combines joint angles and gripper
value into a 5-element array

targetPos = [JointAngles, GripperAngle];

Speed = [50, 50, 50, 50, 50];

arb = Arbotix('port', 'COM4', 'nservos', 5);

arb.setpos(targetPos, Speed);

delete(arb);

clear arb;

end


%PICK AND PLACE: USING COORDINATES



function PickAndPlace(PickCo , PlaceCo)


Speed = [50,50,50,50,50];

CurrentState = 0;


%THE CUBE WITH MINIMUM LENGTH IS 2.0 CM

JawAngle = PositionJaw(2.0);


n = 1;


while n == 1


disp("Current State: " +
num2str(CurrentState));


%THE HOME CONFIGURATION OF THE ARM

if CurrentState == 0

arb = Arbotix('port' , 'COM4' , 'nservos', 5);

arb.setpos([0,0,-pi/4,-pi/2,0],
Speed);

CurrentState = 1;

disp("Current State: " +
num2str(CurrentState));


%FINDING THE JOINT ANGLES TO MOVE
```

```
THE ARM TO PRE GRASP POSE

elseif CurrentState == 1

PickCo(3) = PickCo(3) + 16;

BestSolution1 =
FindOptimalSolution(PickCo(1) ,
PickCo(2) , PickCo(3) , PickCo(4));

CurrentState = 2;



disp("Current State: " +
num2str(CurrentState));



%MOVING THE ARM TO THE PRE GRASP POSE:
HOVER OVER THE CUBE

elseif CurrentState == 2

SetPosition(BestSolution1 , 0);

pause(1);

CurrentState = 3;



disp("Current State: " +
num2str(CurrentState));



%MOVING THE ARM TO GET OVER
THE CUBE BUT DONT GRASP THE CUBE

elseif CurrentState == 3

PickCo(3) = PickCo(3) -13;

BestSolution2 =
FindOptimalSolution(PickCo(1) ,
PickCo(2) , PickCo(3) , PickCo(4));

SetPosition(BestSolution2 , 0);

CurrentState = 4;



disp("Current State: " +
num2str(CurrentState));



%GRASPING THE CUBE

elseif CurrentState == 4

SetPosition(BestSolution2 , JawAngle);

pause(1);

arb = Arbotix('port' , "COM4" , 'nservos' ,5 );

if arb.getpos(5) > 0.9

CurrentState = 5;

else

CurrentState = 4;

end



disp("Current State: " +
num2str(CurrentState));



%FINDING THE JOINT ANGLES TO MOVE THE
ARM ABOVE THE PLACE POSITION

elseif CurrentState == 5

PlaceCo(3) = PlaceCo(3) + 16;

BestSolution3 =
FindOptimalSolution(PlaceCo(1) ,
PlaceCo(2) , PlaceCo(3) , PlaceCo(4));

CurrentState = 6;



disp("Current State: " +
num2str(CurrentState));



%MOVING THE ARM TO HOVER OVER
THE PLACE POSITION

elseif CurrentState == 6

SetPosition(BestSolution3 , JawAngle);
```

```
CurrentState = 7;

disp("Current State: " +
num2str(CurrentState));



%MOVING THE ARM DOWNWARDS AT THE
EXACT PLACE POSITION

elseif CurrentState == 7

PlaceCo(3) = PlaceCo(3) -13;

BestSolution4 =
FindOptimalSolution(PlaceCo(1) ,
PlaceCo(2) , PlaceCo(3) , PlaceCo(4));

SetPosition(BestSolution4 , JawAngle);

CurrentState = 8;



disp("Current State: " +
num2str(CurrentState));



%RELEASING THE CUBE ON THE PLACE POSITION:
LET GO OF THE CUBE BUDDY!

elseif CurrentState == 8

SetPosition(BestSolution4 , 0.8);

CurrentState = 9;

disp("Current State: " +
num2str(CurrentState));



%MOVE UP A LITTLE AFTER PLACING
THE CUBE TO AVOID
COLLISIONS WITH

%OTHER CUBES

elseif CurrentState == 9

PlaceCo(3) = PlaceCo(3) + 12;

BestSolution5 =
FindOptimalSolution(PlaceCo(1) ,
PlaceCo(2) , PlaceCo(3) , PlaceCo(4));

SetPosition(BestSolution5 , 0.8);

n = 0;

end

end

end



%USING THE PIPELINE

%ALL MEASUREMENTS ARE IN CM AND RADIANS



CameraCoords = GetLocationsOfCubes();

WorldCoords = CameraToWorld(CameraCoords);



for i = 1:length(WorldCoords)

Cube = WorldCoords(:,i);

XOffset = 1

if Cube(2) < 0

Cube(1) = sign(Cube(1)) *
(abs(Cube(1)) - XOffset);

else

Cube(1) = Cube(1) + XOffset;

end

Pick = [Cube(1), Cube(2) ,
Cube(3) + 2, -pi/2]

Place = [17 - 1.5 , -8 ,
Cube(3) + 4, -pi/2];

PickAndPlace(Pick , Place);
```

```
end
```