UNIVERSIDAD DE OVIEDO

INTERNATIONAL POSTGRADUATE CENTER

MASTER IN MECHATRONICS ENGINEERING (EU4M)

MASTER'S THESIS

VISION GUIDED 6-AXIS ROBOTIC ARM FOR INSPECTION ON A CONVEYOR LINE

JULY 2024

ALUMNO: BIBEK GUPTA

TUTOR: JUAN CARLOS ÁLVAREZ ÁLVAREZ

UNIVERSIDAD DE OVIEDO

INTERNATIONAL POSTGRADUATE CENTER

MASTER IN MECHATRONICS ENGINEERING (EU4M)

MASTER'S THESIS

VISION GUIDED 6-AXIS ROBOTIC ARM FOR INSPECTION ON A CONVEYOR LINE

JULY 2024

BIBEK GUPTA JUAN CARLOS ÁLVAREZ ÁLVAREZ

ACKNOWLEDGEMENT

I am very grateful to all who have supported me throughout the completion of this Master's thesis.

Firstly, I would like express my deepest gratitude to my supervisor, Prof. Juan Carlos Álvarez Álvarez, Grupo de Investigación SiMuR, at the Universidad de Oviedo, for granting me the opportunity to carry out the Thesis of my interest and access the lab and equipments freely. Many thanks for his valuable guidance and support in formulating and completing my thesis objectives.

I would also like to extend my heartfelt thanks to my academic supervisor Prof. Miguel Ángel José Prieto, Coordinator for Erasmus Mundus Master in Mechatronics for his support and facilitation of procurement of materials required for the conveyor prototype. Additionally, his guidance led me to Prof. José Manuel Sierra Velasco from the Mechanical Department, whose contributions were significant in the design and fabrication of the conveyor.

I am deeply grateful to Prof. José who has been exceptionally helpful and kind. His role in refining and verifying the conveyor design was pivotal. His expertise and assistance were indispensable during the assembly of the prototype. Moreover, his support in accessing the 3D printing lab for essential materials was greatly appreciated.

I also appreciate the assistance of my colleagues and the building staff at the Universidad de Oviedo.

Finally, I extend my gratitude to my friends and family for their unwavering support and encouragement throughout this timeline.

ABSTRACT

This thesis investigates the integration of vision-guided robotic arms in industrial automation and their impact on enhancing operational efficiency, precision, and adaptability. Vision-guided robotics combine advanced imaging systems with robotic capabilities, allowing for real-time environmental interpretation and task execution.

This thesis explains the integration of the UR3e robot with a gripper, camera and a conveyor prototype to study the operational efficiency and precision in industrial automation. The study involved performing and verifying camera calibration and hand-to-eye calibration. A digital twin was created to facilitate the transformation of pixel coordinates from camera images into 3D coordinates relative to the robot, enabling accurate pick-and-place tasks in industrial scenarios.

Three different industrial scenarios were replicated . Firstly detecting the blocks placed on the conveyor and using image processing for inspection based on the colors and sizes. Next an algorithm was developed to detect missing bolts inside the objects on a conveyor line and rejecting them from the conveyor.

KEYWORDS

- Robotics
- Automation
- Image Processing
- Inspection on Conveyor
- Coordinate Transformation
- Computer Vision
- Object Detection
- Mechanical Design
- Quality Control
- Pick and Place
- Camera Calibration
- Hand to Eye Calibration
- Inverse Kinematics

ABBREVIATIONS

- Co-bot: Collabrative Robot
- TCP: Tool Centre Point
- UR3e: Universal Robot Model 3e
- LIDAR: Light Detection and Ranging
- DOF:Degree of Freedom
- IDE: Integrated Development Environment
- VGR: Vision Guided Robot
- CNC: Computer Numerical Control
- RGB: Red Green Blue
- HSV: Hue Saturation Value
- Cmap: Colour map
- BGR: Blue Green Red
- RGI14: Rotary Electric Gripper with Infinite Rotation
- Digital I/O: Digital Input/ Output
- SDK: Software Development Kit
- API: Application Program Interface
- CAD: Computer Aided Design
- TCP/IP: Transmission Control Protocol/ Internet Protocol
- OpenCV: Open Computer Vision library
- .py: Python Script file extension
- .json: JavaScript Object Notation file extension
- DO: Digital Output
- ROI: Region of Interest
- ML: Machine Learning
- AI: Artifical Intelligence
- CNN: Convolution Neural Network

TABLE OF CONTENTS

1 Introduction **1**
1.1 Background 1
1.2 Objectives 1
1.3 Scope of Work 2

2 Literature Review **3**
2.1 Robotic Arms and Vision Systems 3
2.1.1 Importance 3
2.1.2 Process 3
2.1.3 Advantages 4
2.2 Applications of Vision Guided Robots in Industry 4
2.3 Coordinate Systems 6
2.4 Coordinate Systems Transformation 8
2.4.1 Homogeneous Coordinates 8
2.4.2 Coordinates Translation and Rotation 8
2.5 Camera Calibration (Camera Intrinsics) 10
2.5.1 Pinhole Camera Model 10
2.5.2 Distortion 12
2.6 Hand-Eye Calibration (Camera Extrinsics) 14
2.7 Image Processing for Object Recognition: 16

3 System Design and Methodology **19**
3.1 Hardware Components 19
3.1.1 **Universal Robot (UR3e)** 19
3.1.2 RGI14 DH Robotics Gripper 19
3.1.3 Intel Realsense L515 camera 20
3.1.4 Relay 24VDC 21
3.1.5 Conveyor System 22
3.2 Software Tools 23
3.2.1 RoboDK 23
3.2.2 Anaconda and Spyder IDE 23
3.2.3 Solidworks 24
3.2.4 MATLAB 24
3.3 Hardware and Software Integration 25

4 Implementation **27**
4.1 System Setup 27
4.1.1 UR3E Robot communication with PC 27
4.1.2 Digital Twin Setup 27

4.1.3 Connection of RGI14 Gripper with UR3e Robot 29
4.1.4 Connection of Intel RealSense Camera with PC 29
4.2 RoboDK Station Tree 30
4.3 Calibration Procedures 32
4.3.1 Camera Calibration 32
4.3.2 Hand-Eye Calibration 34
4.4 Image Processing Workflow 36
4.5 Pick and Place Workflow 39
4.5.1 Pickpoint Calculation 39
4.5.2 Robot Motion 40
4.6 Pick and Place from Conveyor 41

5 Results and Discussion 43
5.1 Object Detection and Placement Scenarios on Conveyor 43
5.2 Detection based on Red and Yellow Objects 44
5.3 Detection based on Object Size 47
5.4 Detection based on Deficient bolts in the object 48
5.5 Discussions 51
5.5.1 Detection of Rectangular Blocks and Centroid Calculations 51
5.5.2 Detection of coloured objects similar to the background 52
5.5.3 Rejection Based on Deficient Bolts 53
5.6 Challenges Faced 54

6 Conclusion and Future Work 56
6.1 Conclusion 56
6.2 Future Directions 57

7 Appendices 60
7.1 CAD Files 60
7.2 Gantt Chart 65
7.3 Program Codes: 67
7.3.1 Camera_Calib_Acqisition 67
7.3.2 Camera_Calib_Calculation 67
7.3.3 Hand_Eye_Acqisition 69
7.3.4 Hand_Eye_Calculation 70
7.3.5 Main_Program_Conveyor 73

LIST OF FIGURES

2.1 Camera mounted robotic arm 3
2.2 Material Detection and Inspection on a conveyor line utilizing vision robots .. 5
2.3 World Frame with respect to Camera and Object 6
2.4 World Frame with respect to Robot and Object 6
2.5 Image pixel coordinates and normalized coordinates 7
2.6 Translation and Rotation of a 3D coordinate frame 9
2.7 PinHole Camera 3D Projection Model 10
2.8 PinHole Camera Projection Coordinate Frame 11
2.9 Radial and Tangential Distortion 13
2.10 Object grasp coordinates transform flowchart 14
2.11 Hand in Eye and Eye in Hand Setup 14
2.12 Hand-Eye calibration setup 15
2.13 Hand-Eye calibration flow chart 16
2.14 Canny Edge Detection 17
2.15 HSV Cone 17
2.16 Color Segmentation of a Tennis ball 18

3.1 UR3e Robot 19
3.2 RGI14 Gripper 20
3.3 Intel RealSense L515 Camera 20
3.4 Relay 24VDC 21
3.5 Conveyor 3D Model 22
3.6 Sample RoboDK simulation environment 23
3.7 Python IDE 24
3.8 Hardware Integration flowchart 25
3.9 Software Integration flowchart 26

4.1 Connection PC to UR3e (RoboDK Interface) 27
4.2 Digital twin setup for Robot, Gripper and Camera assembly 28
4.3 Objects Classification RoboDk Station Tree 31
4.4 Chessboard Pattern 32
4.5 Sample images for camera calibration 33
4.6 Detected chessboard corners 33
4.7 Mean Projection error for sample images 34
4.8 Sample image for Hand Eye calibration 35
4.9 Flowchart for process of detecting and distinguishing blocks 36
4.10 Image Processing Images Sequence 1 37
4.11 Image Processing Images Sequence 2 38
4.12 Drawn centroid coordinates; color information and object contours for objects of interest 38

4.13 Simulation setup of Robot integrated with conveyor 42

5.1 Real Setup of Integration of Conveyor with Robot 43
5.2 Original and Cropped Image 44
5.3 Image processing steps for Red and yellow colored objects detection from conveyor 45
5.4 Detection Window computations for red and other colored object 46
5.5 a) Original Image b) Cropped Image c) Centroid and Object Detection based on object size 47
5.6 a) Original Image b) Detection Window c) Object Detection and Centroid Calulation 48
5.7 Image processing steps for deficient bolts in objects on the conveyor 50
5.8 a) ROI b) Color Masked c) Bolt Circles Approximation 50
5.9 Detection Window and Centroids for different rectangular blocks 51
5.10 Objects with increased Detection Window and Closely Spaced or Touching ... 52
5.11 Objects with similar Background color 52
5.12 False positive instance during deficient bolts detection 53
5.13 Gripper Jaw Modification 54

LIST OF TABLES

3.1 Conveyor prototype components 22

4.1 Gripper Wiring Setup 29
4.2 Gripper Input Configuration States 29
4.3 Centroids and Targets for detected objects 39

1. Introduction

1.1. Background

Vision-guided robotic systems play a pivotal role in modern industrial automation by combining advanced imaging systems with robotic precision to enhance operational efficiency and accuracy. These robots utilize cameras and sensors to interpret their environment, enabling them to perform complex tasks such as assembly, quality inspection, and material handling with high precision.

At least one robot vision camera will be mounted on the robotic arm itself,literally serving as the eye of the machine. In some cases, additional cameras are installed in strategic locations in the robot's working cell. This set-up allows the camera to have a wider visual angle and capture as much visual data as it needs to perform its function in collaboration with human workers. The robot camera will take 2D or 3D scans of the object. The image will then be stored in the robot database and programmed to trigger the machine to move and perform specific tasks.

1.2. Objectives

Primary:

1. Develop and integrate a vision-guided robotic system for the purpose of inspection and classification.
2. Design and integration of the robotic cell: conveyor system with UR3e robot and sensors.
3. Develop object detection algorithm for classification and pick and place action for randomly placed objects.

Secondary:

1. Optimize camera and hand-eye calibration techniques for accurate operation.
2. Improvement of the Object Detection and Inspection algorithm in a dynamic scenario.

1.3. Scope of Work

This work is conducted in the "Grupo de Investigación SiMuR" laboratory of Universidad de Oviedo. The work aims to upgrade the previous work done on object pick and place applications using UR3E robot. The previous work dealt with picking up small lego bricks and sorting them using YOLO (You only Look Once) detection method.

In this work, a step-wise calibration method of the Intel Real-sense camera and the robot hand to the camera eye is explained. The accuracy of the calibration result are tested in practical environment. Design, fabrication and integration of conveyor is performed with the robot and gripper-camera assembly. A digital twin is developed to perform offline simulation using stored test images.

Next, three relevant industrial scenarios are replicated and object detection algorithm is run on a moving conveyor line with rejections based on object color, size and deficient bolts. All the hardware components and software packages were controlled in Spyder python IDE integrating all in a single program with clear visualizations of results. Identifying challenges encountered during implementation, discussing solutions, and proposing improvements for future implementations are presented.

2. Literature Review

2.1. Robotic Arms and Vision Systems

2.1.1. Importance

Without robotic vision, robots are blind machines that move according to their programming. They rigidly follow the code that dictates their functions, making them ideal for repetitive tasks. They enable defect free production by providing important quality information, such as data about flaws and measurement tolerances, which a blind robot programmed to act within a coordinate system or stage cannot deliver. Now with the arrival of Industry 4.0, robots are also evolving. It allows them to keep up with the demands and trends of the fourth industrial revolution.

Central to the evolution of robotics is the creation of a robot vision system for collaborative robots. Machine or robot vision is a key feature of this evolution, introducing new levels of precision and accuracy in smart automated processes. Vision systems help co-bots perform tasks such as inspecting, identifying, counting, measuring, or reading the bar-code. Ultra-high-speed imaging and lens quality facilitate multi-operations in one process. The ability to perceive their immediate surroundings significantly enhances co-bot capability, which in turn benefits human workers, companies, and industries at large.

Figure 2.1. Camera mounted robotic arm[1]

2.1.2. Process

There are three segments in a robot vision system:

- Image capture: The camera/s will start capturing visual data from a calculated distance. Afterward, the machine will analyze the images or footage and enhance it to produce a clear picture.
- Image processing: The picture will go through further processing and analyzed by pixel. The system will compare the colors and apparent shape of the object with the image programmed in its database.
- Response: Once the machine recognizes that the object in the picture matches the pre-programmed image, it will perform a corresponding action onto the object before it.

2.1.3. Advantages

- Increases efficiency
- Ensures product consistency
- Reliable
- Promotes a safe workplace
- Reduces operating costs

2.2. Applications of Vision Guided Robots in Industry

Vision-guided robots (VGRs) are equipped with cameras and image processing capabilities, allowing them to perform tasks with a high degree of precision and adaptability. They are widely used in various industrial applications due to their ability to enhance productivity, accuracy, and flexibility. Some key applications are listed below:

- **Assembly Line Automation:**
  - Component Placement: VGRs can precisely position components on an assembly line, ensuring correct orientation and placement.
  - Quality Control: They inspect parts for defects, verifying dimensions, alignment, and surface quality.
- **Pick and Place Operations:**
  - Sorting and Packing: Sort and pack products based on size, shape, or color, improving efficiency in packaging processes.
  - Material Handling: They move items between different stages of production, reducing manual handling and associated errors.
- **Inspection and Quality Control:**
  - Non-Destructive Testing: Inspect products for internal and external defects without damaging them, using techniques like X-ray or thermal imaging.
  - Surface Inspection: They detect surface defects such as scratches, dents, or blemishes on products.
- **Welding and Soldering:**
  - Precision Welding: They perform welding tasks with high precision, reducing the risk of defects and ensuring consistent weld quality.
  - Soldering: They handle delicate soldering tasks in electronics manufacturing, where accuracy is critical.

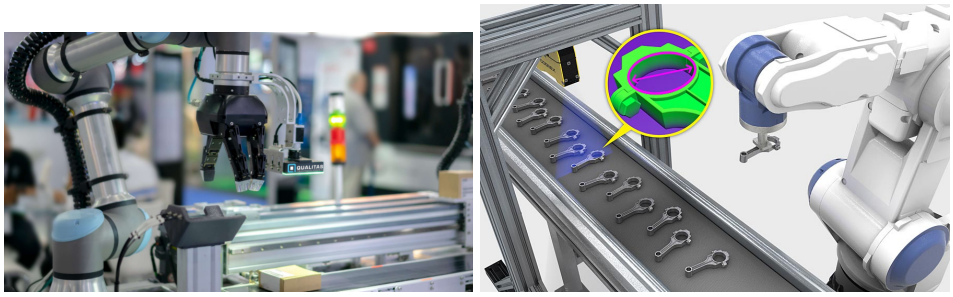


Figure 2.2. Material Detection and Inspection on a conveyor line utilizing vision robots [2]

- **Machine Tending:**
  - CNC Machine Loading/Unloading: VGRs load and unload parts from CNC machines, improving cycle times and reducing operator intervention.
  - Injection Molding: They handle molded parts, removing them from molds and placing them in subsequent processing stages.
- **Palletizing and Depalletizing:**
  - Stacking and Organizing: Stacking products onto pallets and organize them efficiently for storage or shipping.
  - Unloading: They unload products from pallets, preparing them for further processing or distribution.
- **Warehousing and Logistics:**
  - Automated Storage and Retrieval: They retrieve items from storage locations and deliver them to specified areas, optimizing warehouse operations.
  - Inventory Management: They assist in inventory counting and management, providing real-time updates on stock levels.
- **Automotive Industry:**
  - Part Inspection: VGRs inspect automotive parts for defects, ensuring they meet quality standards before assembly.
  - Assembly Assistance: They assist in assembling complex components, such as engines and transmissions, enhancing precision and speed.

2.3. Coordinate Systems

In order to describe the position and orientation, also known as the pose of an object, we define a set of axes and a reference point. The reference point describes the pose of an object in relation to another, with the world usually being treated as the primary frame of reference and providing a global coordinate system for the whole scene.

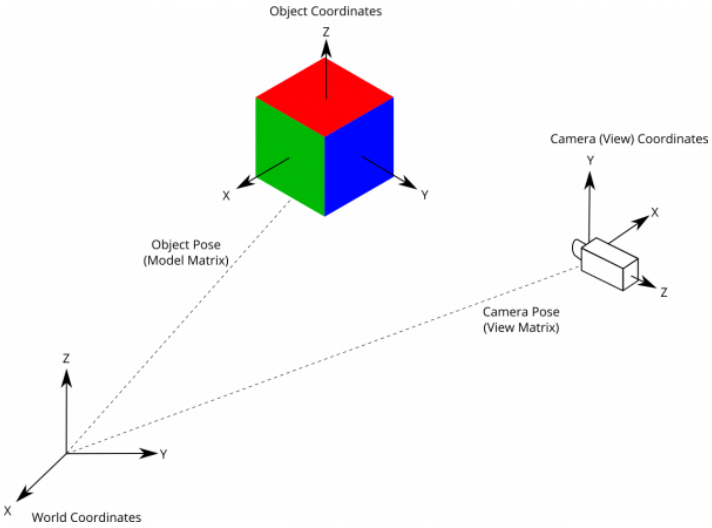


Figure 2.3. World Frame with respect to Camera and Object [3]

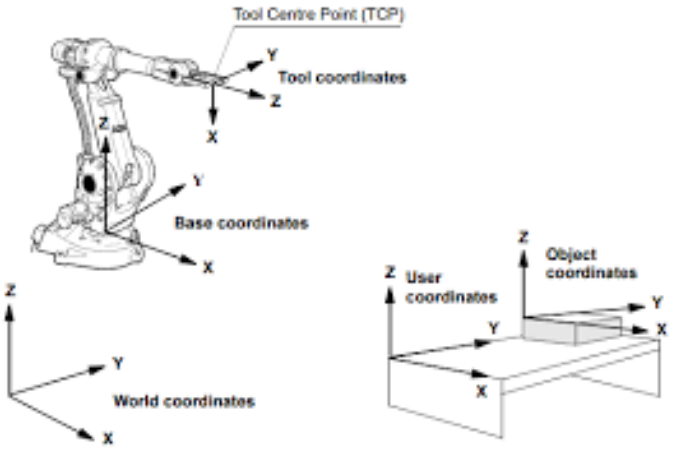


Figure 2.4. World Frame with respect to Robot and Object [4]

In computer vision, for our field of application we define different coordinate systems as:

- **World Coordinate System (3D):** This frames acts as the reference frame for all the other different frames defined in the the workspace. Generally it is known as the origin i.e (0,0,0).
- **Base Coordinate System (3D):** This frame is a fixed Cartesian coordinate frame that represents the center point at the robot's base and defines the overall world for the robot. It specifies the three directions in which the axes are all pointing and the origin position.

- **Tool Frame Coordinate System (3D):** It is defined at the Tool Center Point (TCP). The tool TCP is the point used to move the robot to a Cartesian position. The origin of the TOOL coordinate system is the robot's flange or face plate, the tool's mounting point on axis 6. If no tool is defined, then the point used to move to a Cartesian position is the robot flange itself. A TCP gets generally defined at the center of the tool or the point intended for positioning.
- **Camera Coordinate System (3D):** The camera coordinate frame defines the position of the camera with respect to the robot hand or tool flange eye. This frame is generally found by Hand-Eye calibration which shall be discussed in further details.
- **Object Coordinate System (3D):** The Object Frame is a Cartesian coordinate system that defines the work piece or an area of interest. It could be such as a conveyor, object to grasp or a pallet location.
- **Normalized Image Coordinate System (3D):** The Image coordinates represent the position of a point in the image plane of a camera, normalized with respect to the camera's intrinsic parameters (focal length and optical center). These coordinates provide a way to work with image points independent of the camera's specific characteristics, making them invariant to the camera's zoom factor and principal point.
- **Pixel Coordinate System (2D):** Pixel coordinates refer to the position of a point in an image, measured in pixels. In this coordinate system, the origin (0, 0) is usually located at the top-left corner of the image, with the x-axis increasing to the right and the y-axis increasing downward. The pixel coordinates (u, v) represent the column (u) and row (v) indices of the point in the image. They are directly related to the image's resolution

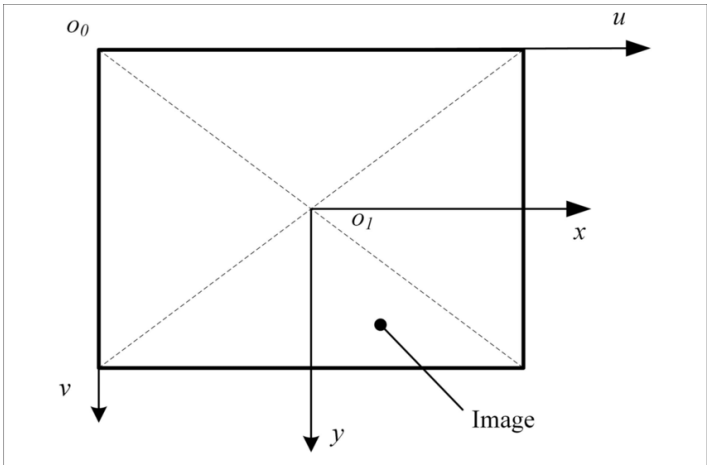


Figure 2.5. Image pixel coordinates and normalized coordinates [5]

2.4. Coordinate Systems Transformation

Coordinate system transformations are fundamental in robotics for tasks like controlling robotic arms, navigating autonomous vehicles, and integrating sensor data. These transformations typically involve rotation and translation, allowing the robot to understand and operate within its environment.

2.4.1. Homogeneous Coordinates

Homogeneous coordinates are an extension of traditional Cartesian coordinates used in projective geometry. They are particularly useful in computer graphics, robotics, and other fields involving transformations and projective spaces. A single matrix can represent affine transformations and projective transformations.

The representation X of a geometric object is homogeneous if X and λX represent the same object for $\lambda \neq 0$. For e.g we have,

$$X = \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} wx \\ wy \\ w \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2.1}$$

In two-dimensional space, a point (x,y) in Cartesian coordinates can be represented as (x,y,1) in homogeneous coordinates. In three-dimensional space, a point (x,y,z) in Cartesian coordinates can be represented as (x,y,z,1) in homogeneous coordinates.

Rotation and scaling transformation matrices only require three columns. But, in order to do translation, the matrices need to have at least four columns. This is why transformations are often 4x4 matrices. However, a matrix with four columns can not be multiplied with a 3D vector, due to the rules of matrix multiplication. A four-column matrix can only be multiplied with a four-element vector, which is why we often use homogeneous 4D vectors instead of 3D vectors.

2.4.2. Coordinates Translation and Rotation

Suppose we wish to translate all points (X, Y, Z) by adding some constant vector (t_x, t_y, t_z) to all coordinates. We can do it by writing out the points (X, Y, Z) as points in R^4 and we do so by taking on a fourth coordinate with value 1. We can then translate by performing a matrix multiplication as:

$$\begin{bmatrix} X+t_x \\ Y+t_y \\ Z+t_z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{2.2}$$

We can perform rotations using a 4×4 matrix as well. Rotations matrices go into the upper-left 3×3 corner of the 4×4 matrix.

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

As shown in the Fig. 2.6 the coordinate frame in the middle is translated to (0,500,0) wrt Reference Frame and the frame on the right is translated by (0,1000,0) as well as rotated by (90,90,45) degrees wrt to the same Reference Frame.

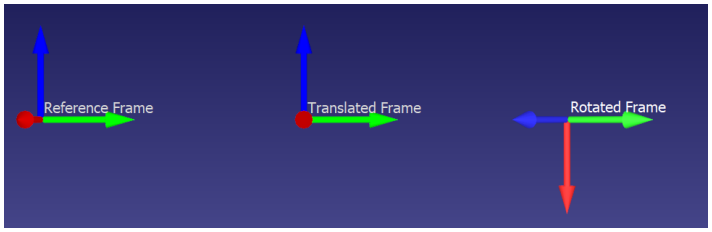


Figure 2.6. Translation and Rotation of a 3D coordinate frame

A general 3D rotation along all axes can be calculated as a product of matrix multiplication given as,

$$R = R_x(\alpha)R_y(\beta)R_z(\gamma) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & -\cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

So, the general form for the homogeneous transformation matrix (T) including combined translation (t) = (t_x, t_y, t_z) and Rotation (R) given in 2.4 can be written as,

$$T = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \tag{2.5}$$

2.5. Camera Calibration (Camera Intrinsics)

Camera calibration is a fundamental task in computer vision crucial in various applications such as 3D reconstruction, object tracking, augmented reality, and image analysis. Accurate calibration ensures precise measurements and reliable analysis by correcting distortions and estimating intrinsic and extrinsic camera parameters.

The process of estimating the parameters of the camera model, consisting of a set of intrinsic, camera model parameters, and a set of extrinsic external parameters, is known as the calibration algorithm. These parameters can be later used to correct the lens distortion, measure lengths in world units of an object or surface, or even it can allow to know the position of the camera into a scene.

2.5.1. Pinhole Camera Model

A pinhole camera is a basic camera without a lens, but with a very small aperture (the pinhole).The light from a scene passes through the aperture and projects an inverted image on the opposite side of the box, which is known as the camera obscura effect.

The aperture is referred to as the pinhole or center of the camera. The distance between the image plane and the pinhole is called the focal length.

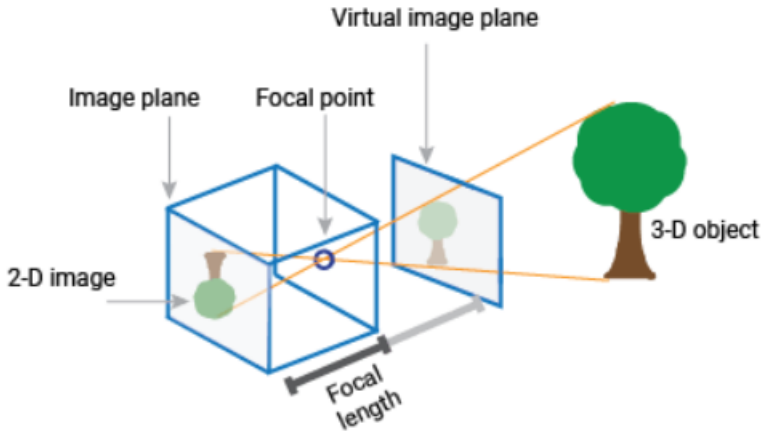


Figure 2.7. PinHole Camera 3D Projection Model [6]

Since the image is inverted we imagine a virtual image plane to revert the image with the same focal length in between the pinhole and the 3D object. Further detailed construction and model representation is discussed in the next Figure.

The figure shows a camera with centre of projection O and the the principal axis parallel to Z-axis. Image plane is at focus and hence focal length f away from O. A 3D point P(X,Y,Z) is imaged on the camera's image plane at coordinate $P_c(u,v)$. Firstly we find the camera calibration matrix C which maps 3D point P to 2D P_c.
Using Law of similar triangle we have,

$$\frac{f}{Z} = \frac{u}{X} = \frac{v}{Y} \tag{2.6}$$

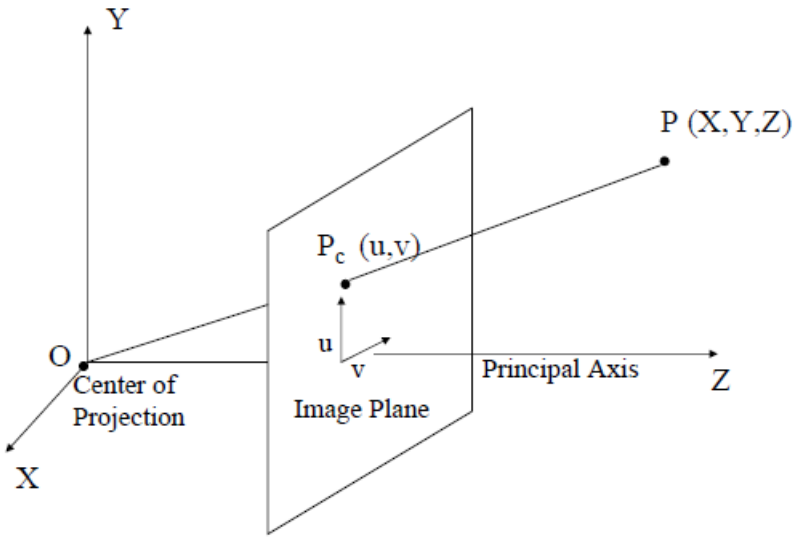


Figure 2.8. PinHole Camera Projection Coordinate Frame [7]

which gives us

$$u = \frac{fX}{Z} \qquad v = \frac{fY}{Z} \tag{2.7}$$

Using homogeneous coordinates for P_c we can write it as

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{2.8}$$

Next, if the origin of the 2D image coordinate system does not coincide with where the Z axis intersects the image plane, we need to translate P_c to the desired origin. Let this translation be defined by (t_u, t_v). Hence, now (u,v) is,

$$u = \frac{fX}{Z} + t_u \qquad v = \frac{fY}{Z} + t_v \tag{2.9}$$

This can be expressed in similar form as 2.8

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} f & 0 & t_u \\ 0 & f & t_v \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{2.10}$$

Since for a camera image we need to express P_c in inches. For this we require the resolution of the camera in pixels/inch. We assume a rectangular resolution with m_u and m_v pixels/inch

in direction u and v respectively. Therefore to measure P_c in pixels, its u and v coordinates should be multiplied by m_u and m_v respectively. Thus,

$$u = m_u \frac{fX}{Z} + m_u t_u \qquad v = m_v \frac{fY}{Z} + m_v t_v \tag{2.11}$$

This can be expressed in matrix form as,

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} m_u f & 0 & m_u t_u \\ 0 & m_v f & m_v t_v \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} P = KP \tag{2.12}$$

Here K only depends on the the intrinsic camera parameters with (f_x, f_y) as point of focus coordinates and (c_x, c_y) as principal or optical axis offset. Sometimes K also has a skew parameter s. This usually comes in if the image coordinate axes u and v are not orthogonal to each other. The Camera Intrinsic matrix is given as,

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.13}$$

Two parameters are currently missing from the formulation: skewness and distortion. It is said that an image is skewed when the camera coordinate system is skewed. In this case, the angle between the two axes are slightly larger or smaller than 90 degrees. Most cameras have zero-skew, but some degree of skewness may occur because of sensor manufacturing errors.

2.5.2. Distortion

Some pinhole cameras introduce significant distortion to images. Two major kinds of distortion are radial distortion and tangential distortion.

Radial distortion causes straight lines to appear curved. Radial distortion becomes larger the farther points are from the center of the image. For example, one image is shown below in which two edges of a chess board are marked with red lines. But, you can see that the border of the chess board is not a straight line and doesn't match with the red line. All the expected straight lines are bulged out.

Radial distortion can be represented as follows:

$$x_{dist} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \quad y_{dist} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{2.14}$$

Similarly, tangential distortion occurs because the image-taking lense is not aligned perfectly parallel to the imaging plane. So, some areas in the image may look nearer than expected. The amount of tangential distortion can be represented as below:

Figure 2.9. Radial and Tangential Distortion [8]

$$x_{dist} = x + [2p_1xy + p_2(r^2 + 2x^2)] \quad y_{dist} = y + [2p_2xy + p_1(r^2 + 2y^2)]$$

Here, r denotes the Euclidean distance between the distorted image point and the distortion center. These five parameters denote the distortion coefficients given below:

Distortion Coefficients=[k_1 k_2 p_1 p_1 k_3]

2.6. Hand-Eye Calibration (Camera Extrinsics)

The task of computing the relative 3D position and orientation between the camera and the robot hand in an eye on-hand configuration, where the camera is rigidly attached to the robot hand, is known as hand-eye calibration. More specifically, this is the task of computing the relative rotation and translation (homogeneous transformation) between two coordinate frames, one centred at the camera lens centre, and the other at the robot hand.

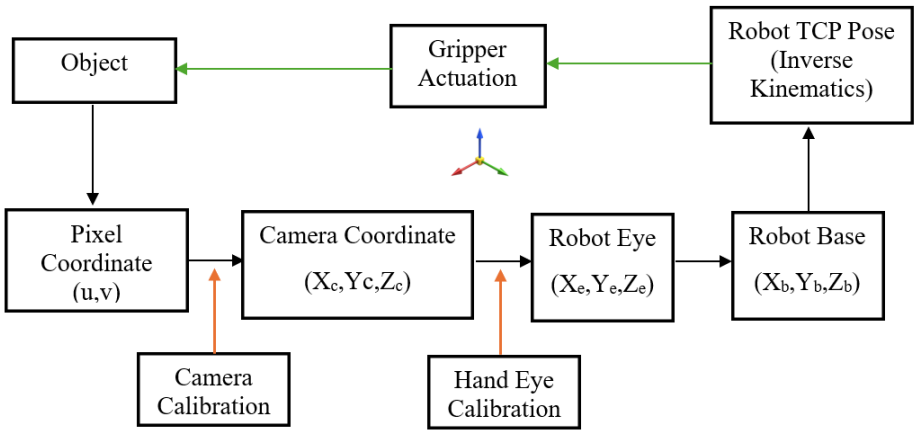


Figure 2.10. Object grasp coordinates transform flowchart

To ensure easy operation of the robot, all commands to the robot are referenced to the robot base frame. Hence, for a complete identification of the object based on the robot base frame, all the relationships must be obtained.

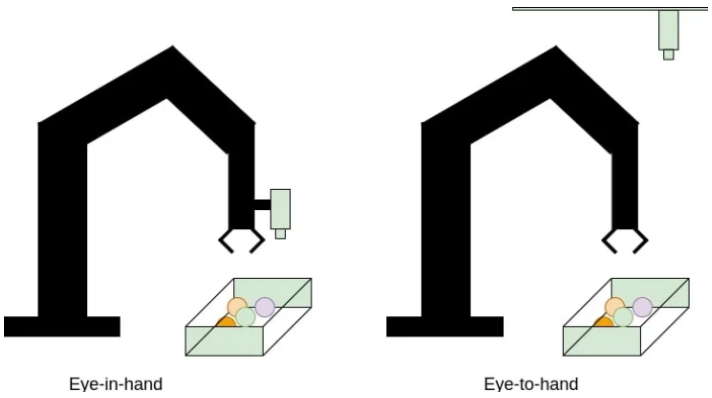


Figure 2.11. Hand in Eye and Eye in Hand Setup [9]

While the relationship between the robot base and the robot hand can be realised from the robot kinematic model, the relationship between the camera and the world can be obtained from camera calibration. This results in the relationship between the camera and the robot hand need to be computed. [10]

The hand-eye transform can be obtained by solving the homogeneous transform equation given by

$$A_{c2}^{c1} \quad X_h^c = X_h^c \quad B_{h2}^{h1} \tag{2.16}$$

where, A_{c2}^{c1} and B_{h2}^{h1} are the homogeneous transform matrices representation of the relative motions of the attached camera and the robot hand between two points, respectively, while X_h^c is the required transform between the robot hand and the camera as shown in 2.12

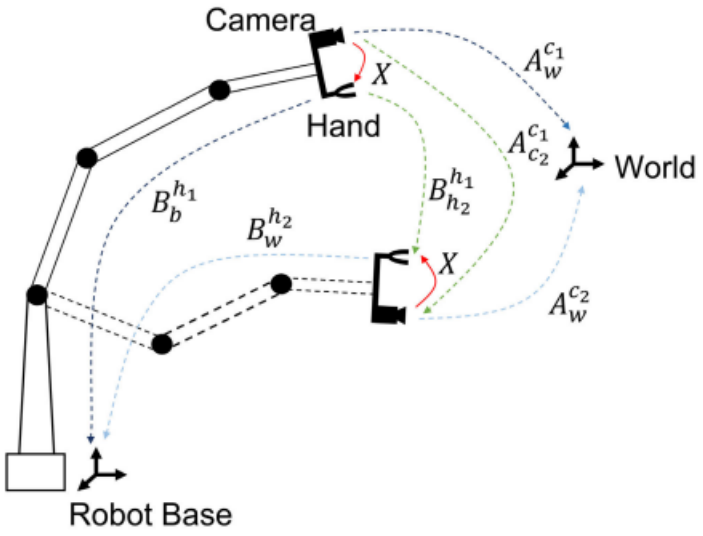


Figure 2.12. Hand-Eye calibration setup [11]

A_{c2}^{c1} and B_{h2}^{h1} can be expressed as the product of two rigid body transform given by

$$A_{c2}^{c1} = A_w^{c1}(A_w^{c2})^{-1} \tag{2.17}$$

and

$$B_{h2}^{h1} = B_b^{h1}(B_b^{h2})^{-1} \tag{2.18}$$

where A_w^{c1}, A_w^{c2} and B_b^{h1}, B_b^{h2} are the poses of the camera with respect to the world frame or calibration object, and the poses of the robot hand and with respect to the robot base respectively, for different robot positions.

2.16, can be represented in matrix form as,

$$\begin{bmatrix} R_A & \vec{t_A} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R_X & \vec{t_X} \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R_X & \vec{t_X} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R_B & \vec{t_B} \\ 0 & 1 \end{bmatrix} \tag{2.19}$$

where R is a 3 × 3 rotation matrix and t is a 3 × 1 translation vector. Hence, the calibration operation involves obtaining sets of robot hand and camera poses as shown in 2.13

While the hand poses can easily be obtained from the robot forward kinematics using the joint encoder readings, the camera pose is usually estimated by observing a set of 3D points provided by a calibration object and their corresponding 2D images (i.e Chessboard or ChArUco-board) using Perspective-n-point algorithm. [12] [13]

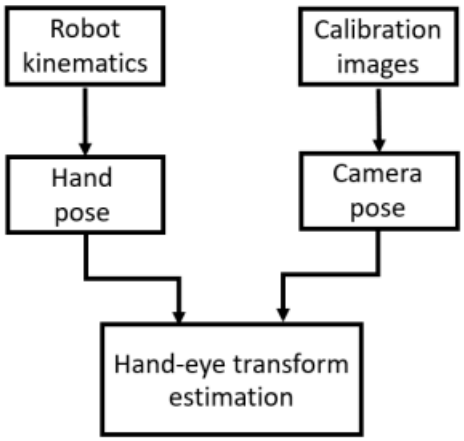


Figure 2.13. Hand-Eye calibration flow chart

2.7. Image Processing for Object Recognition:

The field of computer vision focuses on extracting the meaningful information from images. Image processing, a fundamental aspect of computer vision, includes manipulating and analyzing images to enhance their quality, extract valuable features, and enable automated interpretation. Several techniques are generally used in image processing. Some of these techniques used in this project are discussed below.

- **Filtering and Convolution:** Filtering operations such as blurring, sharpening and noise reduction are applied to images using convolution. Convolution involves sliding a filter or kernel over an image and performing mathematical operations on each pixel. This process enables various improvements such as anti-aliasing, edge detection and texture removal.

  Gaussian blur reduces the noise and detail in an image by averaging the pixel values with their neighbors, with the average weighted more heavily towards the central pixel in a Gaussian manner. It is used in various computer vision algorithms, such as edge detection, to remove small details that might interfere with the algorithm.

- **Edge Detection** Edge detection algorithms identify image boundaries and important transitions. They emphasize areas where the intensity changes quickly, such as edges, curves or contours. Edge detection plays an important role in object detection, shape analysis and feature extraction. Popular edge detection algorithms are Canny edge detector and Sobel operator.

  For our application we use canny edge detector. Since edge detection is susceptible to noise in the image, first step is to remove the noise in the image with a 5x5 Gaussian filter. Smoothened image is then filtered with a Sobel kernel in both horizontal and vertical direction to get first derivative in horizontal direction (G_x) and vertical direction (G_y). From these two images, we can find edge gradient and direction for each pixel as follows:

$$Edge \quad Gradient(G) = \sqrt{{G_x}^2 + {G_y}^2} \quad and \quad Angle(\theta) = \tan^{-1}(G_y/G_x) \tag{2.20}$$

Gradient direction is always perpendicular to edges. It is rounded to one of four angles representing vertical, horizontal and two diagonal directions.

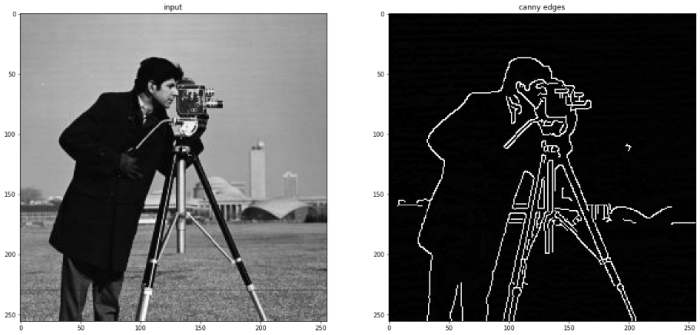


Figure 2.14. Canny Edge Detection [14]

- **Colour Scales and Conversion:** Color scale conversion in image processing involves transforming images from one color space to another, which is crucial for various applications like image enhancement, segmentation, and analysis.

  1. **RGB** is a color model that represents colors as mixtures of three underlying components — red, green, and blue color channels — that create various hues when combined.Each color is a triplet (R, G, B) where each component can range from 0 to 255.

     Grayscale: It represents shades of gray with intensity ranging from 0 (black) to 255 (white).

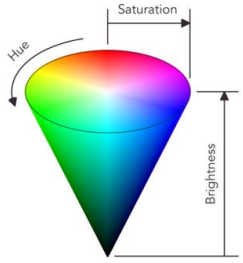


Figure 2.15. HSV Cone [15]

  2. **HSV:** In this model The H stands for Hue, S stands for Saturation, and the V stand for value. Hue is the color portion of the model, expressed as a number from 0 to 360 degrees. For. eg.
     Red falls between 0 and 60 degrees.Yellow falls between 61 and 120 degrees.Green falls between 121 and 180 degrees. Blue falls between 241 and 300 degrees.

     Saturation describes the amount of gray in a particular color, from 0 to 100 percent. Reducing this component toward zero introduces more gray and produces a faded effect.

     Value works in conjunction with saturation and describes the brightness or intensity of the color, from 0 to 100 percent, where 0 is completely black, and 100

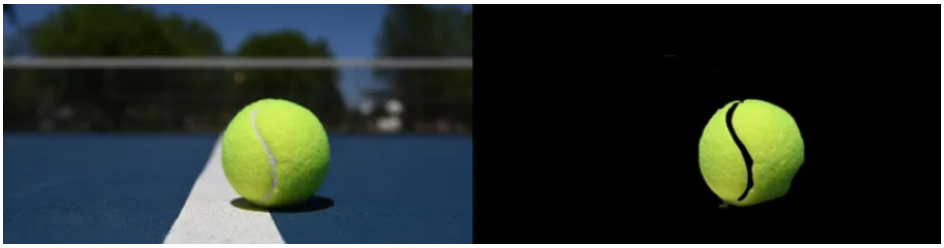


Figure 2.16. Color Segmentation of a Tennis ball [16]

is the brightest and reveals the most color.

3. **RGB to Grayscale**: This is done by lumosity method which takes into account the human perception by giving different weights to the RGB components.

$$Gray = 0.299R + 0.587G + 0.114B \tag{2.21}$$

4. **RGB to BGR:** This conversion swaps the pixel color values for the (R,G,B) channel to (B,G,R). For. e.g if [240, 26, 0] is represented in RGB, its conversion to BGR shall be [0, 26, 240] and vice versa.

5. **RGB to HSV:** To get the most out of color masking for image segmentation, we convert the RGB image to HSV model. RGB color space is best suited for images with simple colors, while the HSV color space is better for images with complex colors. This shall be discussed in the next section.

- **Image Segmentation:**

1. **Thresholding:** It is the simplest method of segmenting images. Here, we convert an image from colour or grayscale into a binary image, i.e., one that is simply black and white. Most frequently, we use thresholding as a way to select areas of interest of an image, while ignoring the parts we are not concerned with. We blur the image and create a binary mask with values of 0 and 1. This binary mask is then applied to the original colored image to extract only the region of interest.
2. **Color Masking:** With the image in a suitable color space, the next step is to create the color mask. This involves defining a range of colors in the image that we want to extract and then converting the image into a binary format where the pixels that fall within the specified color range are set to 1, and the pixels that do not fall within the specified range are set to 0. the color mask is applied to the image by multiplying the image by the binary mask. This results in an image that only includes the objects or parts of the image that correspond to the specified color range.post processing may involve adjusting the color range or modifying the binary mask to better extract the desired objects or parts of the image.

3. System Design and Methodology

3.1. Hardware Components

3.1.1. Universal Robot (UR3e)

The Universal Robot UR3e is an ultra-light, compact collaborative industrial robot ideal for table-top applications. Its small footprint makes it ideal for building directly into machines or other tight work spaces. This co-bot weighs 11 kg, and has a payload capacity of 3 kg. The ±360-degree rotation of all wrist joints and infinite rotation of the end joint make it well suited for light assembly and screwing tasks.

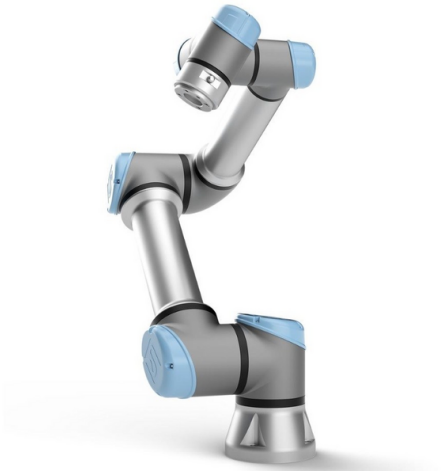


Figure 3.1. UR3e Robot [17]

Basic specifications: [17]
Reach: 500mm
DOF : 6 rotating Joints
Axis: Base, Shoulder, Elbow, Wrist 1, Wrist 2, Wrist 3
Footprint: $\phi 128mm$
Applications: Machine tending, Quality Inspection, Pick and Place, Assembly (with inclusion of feedback force sensor), simple collaborative applications such as picking and bringing the tool to the user.

3.1.2. RGI14 DH Robotics Gripper

The RGI14 gripper is a robotics gripper by DH robotics. It offers a pararallel jaw design, where two jaws move in parallel to each other, and infinite rotation of fingers. For rotation and gripping tasks, there are 2 sets of servo motors and drive controller inside the housing. The communication Interface is via RS485 Interface and Digital I/O.

Basic specifications: [18]
Grippping Force: 10-35N
Stroke: 14mm
Maximum Load Capacity: 0.7 kg
Opening/Closing: 0.3s/0.3s

Figure 3.2. RGI14 Gripper [18]

Nominal Voltage: 24VDC

3.1.3. Intel Realsense L515 camera

The Intel RealSense L515 is a compact and advanced LiDAR camera designed for high-precision depth sensing and 3D scanning. It utilizes Time of Flight (ToF) to capture depth information which makes it suitable for a wide range of robotics, augmented reality, virtual reality and computer vision applications. The camera boasts impressive depth accuracy, with less than 1 cm error at a range of up to 9 meters, making it suitable for applications requiring precise distance measurements. It uses a USB-C connection for data transfer and power, ensuring fast and reliable connectivity with compatible devices. It is supported by the Intel RealSense SDK 2.0, which provides tools and libraries for easy integration and development, including support for multiple operating systems like Windows, Linux, and macOS. The camera is compatible with programming languages such as Python, MATLAB, and Robot Operating System (ROS).

Figure 3.3. Intel RealSense L515 Camera [19]

Basic specifications: [19]
RGB frame resolution: 1920 × 1080
RGB sensor FOV (H × V):70° × 43° (±3°)
RGB sensor resolution: 2MP
Frame rate: 30fps
Minimum depth distance (Min-Z) at max resolution: 25 cm
Depth Accuracy: 5 mm to 14 mm thru $9m^2$
Depth Field of View (FOV): 70° × 55° (±3°)
Depth output resolution: Up to 1024 × 768

3.1.4. Relay 24VDC

A relay is an electromechanical device that uses an electrical coil to control the opening and closing of contacts in another circuit. The coil voltage required to activate the relay is 24 VDC, meaning that a direct current voltage of 24 volts is needed to energize the coil, causing the contacts to either close or open depending on the relay type. This configuration makes it suitable for various applications where medium power switching is necessary, such as in industrial automation systems, household appliances, and automotive electronics, providing reliable isolation and control between low power control circuits and higher power load circuits.

Specifications:
Contact capacity: 5 A, 250 VAC/30 VDC
Coil voltage: 24 VDC

Figure 3.4. Relay 24VDC [20]

3.1.5. Conveyor System

The Conveyor System is used to transport the object from one point to the another. For the application in this work, a prototype of the conveyor was designed to transport the objects and apply the pick place detection algorithm.

The conveyor is driven by a 500 Watts DC powered 2202VAC motor actuated by 24V DC relay for object of dimension (lxbxh) as (75x25x15)mm. The components and their specifications are shown in the Table 3.1. The model was designed and assembled in Solidworks a shown in Fig. 3.5

S.N.	Materials	Quantity	Specifications
1	Timing Belt Pulley	6	2GT 20 Teeth 8mm Bore
2	Closed Timing Belt	3	752GT; Width: 6mm
3	Bearing Block	4	608ZZ
4	Flexible Coupling	1	8mm Bore
5	Aluminium Shaft	2	Diameter: 8mm; Length: 200mm
6	Relay Block	1	24V DC Coil/ Contact Points: 220V AC
7	Motor	1	220VAC; 50rpm
8	Aluminium Slotted Profiles	2mtr.	30mm X 30mm
9	Frame Connections	10	30X30 8mm Slot Brackets

Table 3.1. Conveyor prototype components

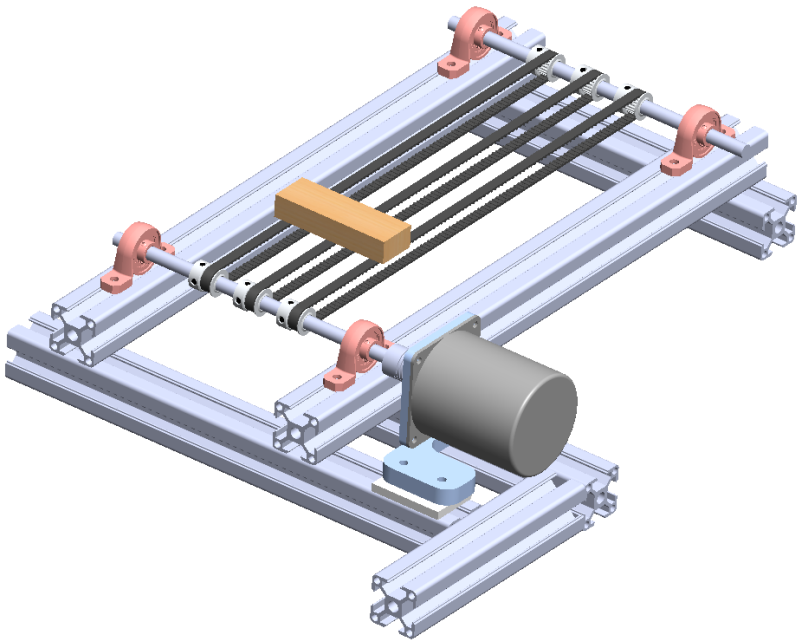


Figure 3.5. Conveyor 3D Model

3.2. Software Tools

3.2.1. RoboDK

RoboDK is a versatile and powerful software platform designed for robotics and industrial automation applications.RoboDK provides a simulation and programming environment for industrial robots, making it easier for users to create, simulate, and deploy robotic applications. It supports a wide range of robots from different manufacturers and can be used for various tasks, including robotic machining, 3D printing, pick and place, etc. Here we

Figure 3.6. Sample RoboDK simulation environment [21]

can program robots offline, i.e creation and testing programs without the need to physically access the robot. This saves time and allows for safer programming environments. The software features collision detection to ensure that the programmed paths do not cause the robot to collide with objects in its environment. RoboDK supports Python scripting, providing users with the ability to create custom scripts for specific tasks.

RoboDK features a user-friendly interface that makes it accessible to both novice and experienced users. It includes drag-and-drop functionality, making it easy to build and modify simulations. The software supports plugins for various CAD/CAM software, including SolidWorks allowing for seamless integration and workflow.

The RoboDk API [22] for python helps to integrates all the offline programming features of the RoboDK simulator and allows to deploy automated applications for a large variety of robots and mechanisms.

Details:
Version: RoboDK v5.2.5 (64 bit) for Windows
Used License: Educational (Universidad de Oviedo)

3.2.2. Anaconda and Spyder IDE

Anaconda is a popular open-source distribution of Python and R programming languages for scientific computing, data science, machine learning, and data analytics. It simplifies package management and deployment and includes a wide range of data science packages.

Anaconda comes with Conda, a powerful package manager that allows to install, update, and manage libraries and dependencies for your projects easily. It allows to create isolated

environments to avoid conflicts between different project dependencies. Spyder (Scientific

Figure 3.7. Python IDE [23]

Python Development Environment) is an open-source integrated development environment (IDE) included in the Anaconda distribution [23]. It is specifically designed for scientific programming in Python.

It has an interactive python console that allows us to view and edit variables, numpy arrays, other data types, etc. It also has a built in support for displazing plots and graphics generated by libraries like Matplotlib. A robust debugging tool that allows you to set breakpoints, inspect variables, and step through code.

Details:
Conda Version: 24.1.2
Spyder IDE Version: 5.5.1
Python version: 3.11.9

3.2.3. Solidworks

SolidWorks is a powerful computer-aided design (CAD) software program used for 3D modeling and design, primarily in the fields of mechanical engineering and product design. Developed by Dassault Systèmes, SolidWorks is widely recognized for its ease of use, robust features, and comprehensive toolset, making it a popular choice among engineers, designers, and manufacturers.
Version: SolidWorks 2022 (Student Version)

3.2.4. MATLAB

MATLAB (Matrix Laboratory) is a high-level programming environment and software platform developed by MathWorks, primarily used for numerical computing, data analysis, algorithm development, and visualization. It is widely utilized in engineering, science, economics, and academia for its robust functionality and ease of use.
Version: MATLAB 2024a (Student License)

3.3. Hardware and Software Integration

- **Hardware Integration:**
  The presented flowchart outlines an integrated robotic automation system featuring a UR3e robot, a PC, a camera, a gripper, a relay, and a conveyor. The camera captures visual data and sends it to the PC via USB 3.2 for processing. Based on the analyzed data, the PC communicates with the UR3e robot through TCP/IP, instructing it to perform specific tasks. The UR3e robot, equipped with a gripper connected via a Lumberg RKMV 8-354 connector, executes precise object manipulation tasks. Additionally, the robot controls a relay using a 24V DC signal to manage the operation of a 220V AC-powered conveyor. This cohesive system allows for efficient automation of object handling and movement processes, with the PC serving as the central control unit coordinating the actions of all components.

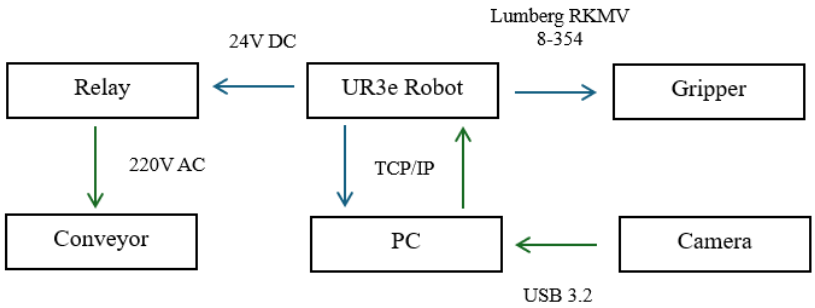


Figure 3.8. Hardware Integration flowchart

- **Software Integration:**
  The diagram illustrates a software integration workflow for controlling a UR3e robot using RoboDK and Python programming. The main program, developed in the SPYDER Python IDE, integrates with RoboDK Station for robot simulation and control. It utilizes the Intel RealSense L515 Camera SDK 2.0 for vision processing. The UR3e driver interfaces with the UR3e robot, providing real-time feedback on the robot's pose and handling digital I/O signals. These signals control the gripper's actions (open, close, rotate) and the conveyor's operation (start, stop). The flow of information is bi-directional between the main program and the RoboDK Station, and between the UR3e driver and the UR3e robot, ensuring synchronized and precise control of the robotic system.

  RoboDK provides two Python libraries, RoboLink and RoboMath, which are essential for creating and managing robot simulations, programs, and mathematical operations.

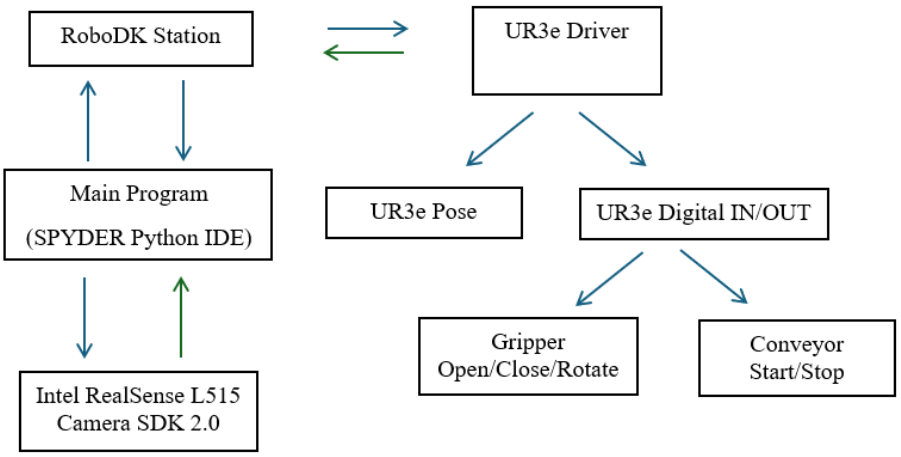


Figure 3.9. Software Integration flowchart

- **Robolink.py:**
  RoboLink is the main interface to communicate with RoboDK's API. It allows you to create, modify, and control objects in the RoboDK environment, such as robots, tools, and targets. It also provides functions to load and save projects, manage connections, and interact with the RoboDK station.

- **Robomath.py:**
  RoboMath provides mathematical functions for working with robotics-specific transformations and operations. It includes utilities for handling homogeneous transformations, Euler angles, quaternions, and other common robotics calculations. This toolbox has been inspired from Peter Corke's Robotics Toolbox. [24]

4. Implementation

4.1. System Setup

4.1.1. UR3E Robot communication with PC

The connection between UR3E robot and PC can be established in various ways, for e.g. TCP/IP, Real Time Data Exchange (RTDE), Dashboard server and XML-RPC(Remote Procedure Calls) library for complex interactions. For the project application, we send URScipt over TCP/IP and recieve feedback in real time. [25]

To achieve the communication, the PC and the robot was connected to the same network. Next, The TCP/IP socket connection on the robot with below details was set up to send/receive URScript via RoboDK connection interface.

HOST: 156.35.152.57 (Robot IP)
PORT: 30002 (Default Port for URScript)
Local IPv4 Address: 156.35.152.57 (PC IP)

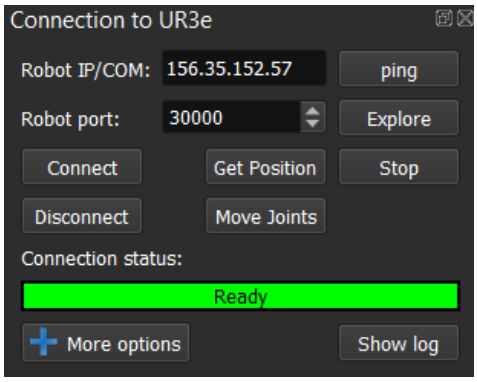


Figure 4.1. Connection PC to UR3e (RoboDK Interface)

4.1.2. Digital Twin Setup

Setting up a digital twin in RoboDK involves creating a virtual model of your robot and its environment. This model is used to simulate, program, and optimize the robot's tasks before deploying them to the actual robot. RoboDK supports many robot manufacturers, and robot models can be downloaded directly from the RoboDK library. The CAD Models of the Gripper and Camera was assembled in Solidworks and imported to the RoboDk Station environment.

To link the CAD models with each other, different frames of reference were defined the position and orientation (X,Y,Z,u,v,w) of different objects was defined to replicate the actual dimensional settings. The details of Position (X,Y,Z) and Orientation (u,v,w) of different frames and components linked for the Digital twin is mentioned in detail below :

Coordinate Frames:

- World Frame: (0,0,0,0,0,0)
- UR3e Base Frame (w.r.t World Frame): (0,0,15,0,0,0)
- Robot Flange Eye (w.r.t World Frame) : (305.3, -131.05, 480.8, -69.28, 69.28, -69.28)
- Gripper Tool Centre Point (TCP) (w.r.t Robot Flange Eye) : (0,145,35,0,0,0)
- Camera Eye Frame (w.r.t Robot Flange Eye): (-1.766, 103.341, 124.898, -89.484, 0.516 , 0.286)
  From Equation: 4.3

Components:

- Mounting Base Plate (wrt World Frame): (0,0,0,0,0,0)
- Table (1400 X 800 X 800)mm (wrt World Frame): (250,-250,0,0,0,0)
- Gripper Camera Assembly (w.r.t Robot Flange Eye): (0,0,0,0,0,0)
- Conveyor Assembly (wrt World Frame: (210,175,-220,90,0,0)

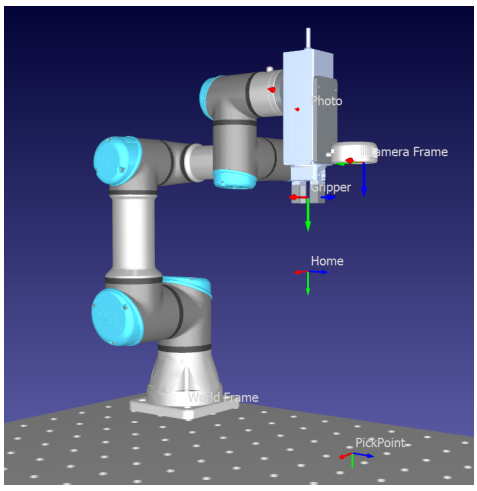


Figure 4.2. Digital twin setup for Robot, Gripper and Camera assembly

4.1.3. Connection of RGI14 Gripper with UR3e Robot

The eight power and signal pins of the gripper are connected to the eight pins connector of the UR3e via a Lumberg RKMV 8-354 cable. Table 4.1 The power lines and the digital IN/OUT pins are only used for this project. The tool voltage output was set as 24V in the settings of teach pendant.

	UR3E Pins		Gripper Wires	
S.N.	Color	Description	Color	Description
1	Red	GND	Black	GND
2	Gray	24V +	Red	24V
3	Blue	DO 0	White	DI 1
4	Pink	DO 1	Brown	DI 2
5	Yellow	DI 0	Yellow	D0 1
6	Green	DI 1	Orange	D0 2
7	White	RS485+ 0	Green	RS485 A
8	Brown	RS485- 0	Blue	RS485 B

Table 4.1. Gripper Wiring Setup

Gripper Settings:

- Gripper position: 0 (fully closed) and 1000 (fully open)
- Gripper Speed: 100% (both opening and closing)
- Gripper Force: 100 % (Opening) and 50% (closing)

The gripper is programmed to run in Digital I/O mode with two binary bits resulting in four different configurations.

Digital IN 1	Digital IN 2	Description
0	0	Gripper Closed
0	1	Gripper Closed and Rotate 90° (Vertical)
1	0	Gripper Open
1	1	Gripper Open and Rotate 90° (Horizontal)

Table 4.2. Gripper Input Configuration States

4.1.4. Connection of Intel RealSense Camera with PC

The L515 camera is connected to the PC via USB 3.2 cable to send and receive Information. The camera has a windows application named as " Intel RealSense Viewer" which helps to visualize the image and depth stream as well as configure different settings such as Exposure, Brightness, Contrast, HSV etc. However for our application we used the Intel RealSense SDK 2.0 package to receive image streams as per requirement settings.

The image and depth streams are streamed at (640,480) resolution. The image is captured and saved when the letter 's' on the keyboard in pressed and stream is terminated with the press of letter 'q'.

Reference code file: *Camera_Start.py*

4.2. RoboDK Station Tree

Before discussing the RoboDk station tree in detail, the different components of the RoboDk environment are explained in brief below:

- **Station:** A station represents the entire setup of the robotic cell or work environment. It includes all the elements necessary for the simulation, such as robots, tools, parts, fixtures, and other devices. The station serves as the workspace where the interactions between these elements are defined and managed.
- **Targets:** Targets are specific points in space that a robot needs to move to or interact with. These points are defined in the station and can include various parameters such as position, orientation, speed, and other motion settings. Targets are crucial for creating precise and repeatable robot movements. They can be defined in different reference frames and can be adjusted or fine-tuned as needed.
- **Program:** A program in RoboDK is a sequence of instructions that tells the robot how to move and what actions to perform. Programs are created by linking targets and defining the robot's path and operations between these targets. Programs can include linear and joint movements, tool activations, loops, conditional statements, and other control structures. It allows to simulate these programs to ensure correct operation before deploying them to the actual robot.

  For the execution of the task, the targets and programs placd in the environment is shown in Fig. 4.3 and explained below. All Target coordinates are with respect to the world frame.

  The targets used in the station tree are explained below:

  1. **Image Capture:** Moves the robot to image capture position.
     Coordinates:
     (340.299957, -108.778937, 458.204773, -69.282000, 69.282000, -69.282000)
  2. **Home:** After object centroid calculation, the TCP moves to the home target position. Checks for the orientation of the object to correct the gripper jaw rotation and moves to the next target.
     Coordinates:
     (340.300000, -131.050000, 250.000000, -69.282032, 69.282032, -69.282032)
  3. **Pickpoint:** This is the final coordinate obtained from pixel coordinate transformation to world frame. This is calculated for each object of interest after successful pick and place operations so it is a dynamic target coordinate calculated

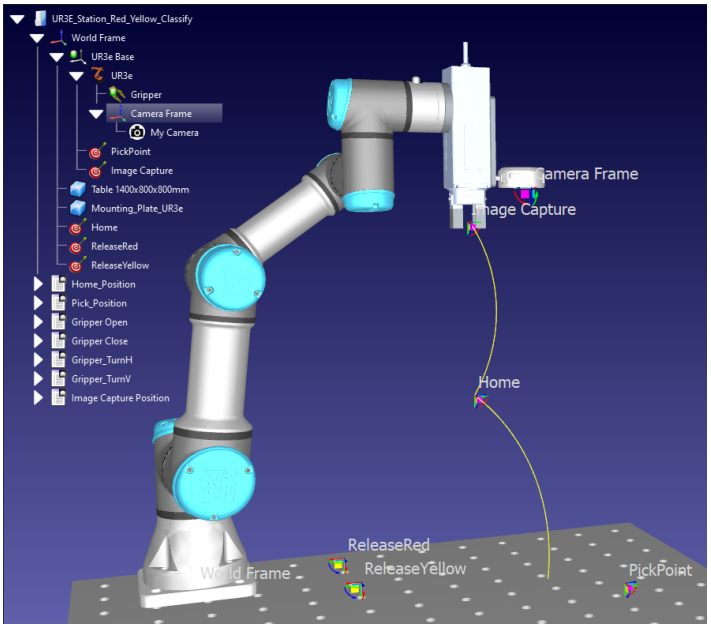


Figure 4.3. Objects Classification RoboDk Station Tree

by the main program function target_set().
Reference code file: *Main_Program.py*

4. **Release Red:** After grasping the red object the TCP moves back to Home target and according to the color classification moves to the designated red rack drop coordinate.
   Coordinates:
   (340.300000, -131.050000, 250.000000, -69.282032, 69.282032, -69.282032)
5. **Release Yellow:** After grasping the yellow object the TCP moves back to Home target and according to the color classification moves to the designated yellow rack drop coordinate next to the redone..
   Coordinates:
   (135.804834, -431.388519, 110.891531, -2.136588, -126.292073, 125.811757)

The movement programs used in the station tree are explained below: All the positional movement program use movej() function, provided the target coordinates, speed and acceleration as function arguments.
for.e.g.
movej(X, Y, Z, u, v, w, acceljoints, speedjoints)

1. **Home_Position:** Moves the TCP to Home target coordinate.
2. **Pick_Position:** Moves the TCP to calculated target Centroid coordinate.
3. **Image Capture Position:** Moves the TCP to Image capture target coordinate.
4. **Gripper Open:** Actuates the Gripper in fully open position.
5. **Gripper Close:** Actuates the Gripper in fully Closed position.
6. **Gripper_TurnH:** Rotates the Gripper in Horizontal Jaw orientation.
7. **Gripper_TurnV:** Rotates the Gripper in Vertical Jaw orientation.

The speed and acceleration for the joints are defined as: (for Pickpoint target program)

Joints Speed: $25^\circ/s$
Joints Acceleration: $70^\circ/s^2$ The speed and acceleration for the joints are defined as: (for all other target programs)

Joints Speed: $50^\circ/s$
Joints Acceleration: $150^\circ/s^2$

The 50% reduced speed is a safety precautionary as not to damage the gripper jaw and the objects during approaching the object.

4.3. Calibration Procedures

4.3.1. Camera Calibration

To start with the camera calibration, a step by step procedure was adopted as explained below:

- **Camera Mounting:** The Camera was mounted onto the gripper with a help of mounting plate and secured such as the eye of the camera faces vertically downwards exactly at 90deg. To continue further the camera position should be kept fixed. After the gripper and camera were assembled together with the robot tool flange, the robot was moved to a specific pose, which defines the Image taking position for the calculations to be performed ahead.
- **Chessboard Pattern:** A 9 X 6 chessboard pattern was printed on an A4 size paper with each box measuring 24mm exactly. This pattern was kept below the camera field of view assuring good lighting to capture clear images of the chessboard.

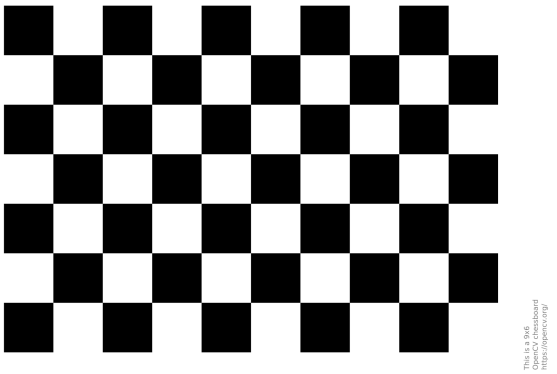


Figure 4.4. Chessboard Pattern [26]

- **Capture Images:** Multiple images of the chessboard were taken from different angles and positions ensuring that the chessboard fills the image frame and is clearly visible. A total of 10 images were recorded.
  Reference code file: ***Camera_Calib_Acquisition.py***

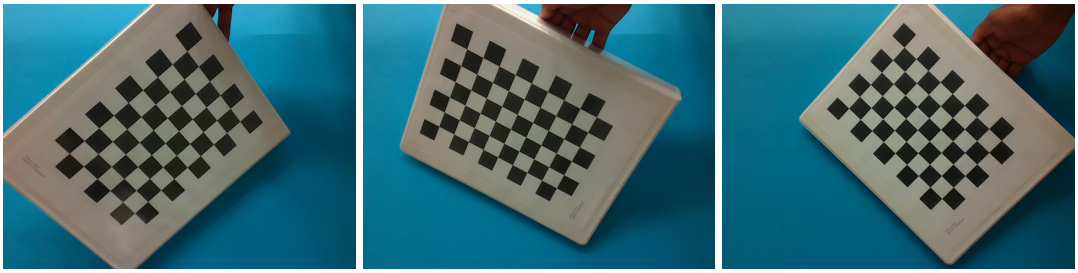


Figure 4.5. Sample images for camera calibration

- **Loading Images and Corner Detection** The chessboard corners were detected by loading the camera images into OpenCV library environment in Spyder IDE. The chessboard pattern size was defined. The chessboard is detected using the function findChessboardCorners(). Here we define the 3D object points in the 3D space and image points which are on the 2D image plane with the help of cornerSubPix() function.

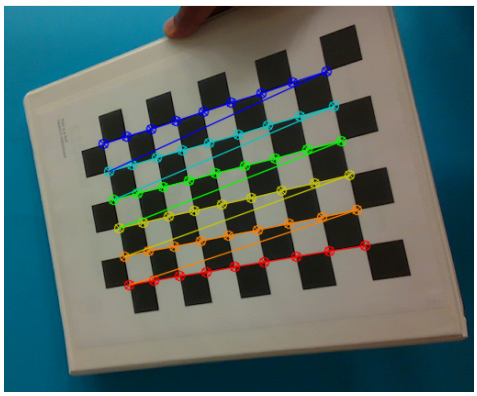


Figure 4.6. Detected chessboard corners

- **Compute Calibration matrix:**

  The object and image points are passed to the openCV library function calibrateCamera() which return us the calibration matrix, reprojection error and distortion coefficients.
  Reference code file: ***Camera_Calib_Calculation.py***

  The camera intrinsics matrix was found to be:

$$K = \begin{bmatrix} 599.51 & 0 & 328.63 \\ 0 & 600.228 & 247.71 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.1}$$

  Overall RMS re-projection error: 0.12
  Distortion Coefficients: [0.04, 0, 0, 0, 0.24]

  Corrected Camera matrix after accounting distortion:

$$K = \begin{bmatrix} 599.53 & 0 & 331.04 \\ 0 & 600.37 & 246.21 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

- **Evaluation of calibration Accuracy:** Reprojection error is a measure of the accuracy of the camera calibration process. It quantifies how well the estimated camera parameters (intrinsic and extrinsic) map the 3D points in the world to the 2D points in the image plane.

  We start with known 3D points in the world and their corresponding 2D points in the image. Using the estimated camera parameters, the 3D points are projected into the image plane to get the predicted 2D points.The difference between these points is the re projection error.Low Re projection Error indicates that the estimated camera parameters are accurately mapping the 3D points to the 2D image points, implying good calibration. [27]

  The average re projection error was found to be : 0.12

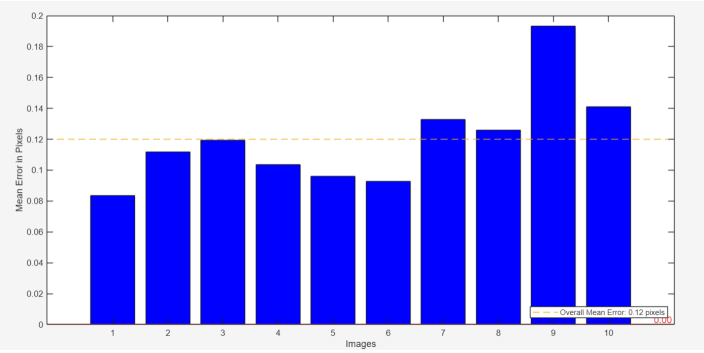


Figure 4.7. Mean Projection error for sample images

4.3.2. Hand-Eye Calibration

Here the position of the camera relative to the robot tool flange eye is calculated. The steps taken to perform Hand Eye calibration is discussed below:

- **Design of a Digital Twin:** Firstly, a digital twin of the robot, gripper and camera is created in RoboDk station. The mounting table and flange are assembled with the robot. Next the gripper and camera are assembled to the tool flange eye with the help of additional mounting plates accurately with real setup dimensions.

  Using the IP address of the robot, the connection between RoboDK and the robot is established through a common network.

- **Camera and Robot Setup:** In the first step the assembly of the robot with gripper and camera is secured in a fixed position. Next the calibration object i.e the chessboard pattern is Placed on the table such that the camera field of vision captured the entire pattern. It is important here to keep the calibration object at a fixed position.

- **Image Capture and Recording Robot Pose:**

  The Robot was controlled in local mode using Polyscope. Free drive operation was enabled to move the robot arm by manually to different positions ensuring the chessboard pattern was clearly visible at each different positions. The robot poses were recorded in the digital twin environment as targets.

The robot was moved to each defined target and images of the chessboard was taken synchronized with recording of the robot joint angles and position and orientation of the tool flange eye as a .json file. A total of 8 sample images and corresponding poses were recorded for this calibration process.
Reference code file: ***Hand_Eye_Acquisition.py***

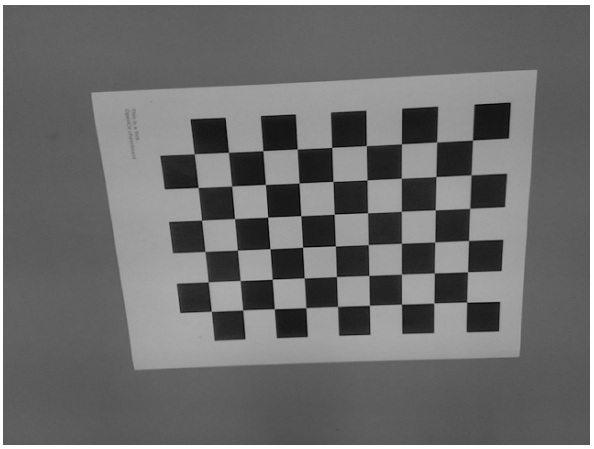


Figure 4.8. Sample image for Hand Eye calibration

The corresponding recorded Robot joint angles and flange eye for the 4.8 is as:

Joints angles: [-0.775, -129.064, -20.313, -47.799, 88.718, 0.953]

Flange Eye Pose: [448.288, -139.188, 503.946, 3.022, 1.268, 1.705]

Here the joint angles respresent the angles of the six linking arms and the for the flange eze pose the first three represent the position (X,Y,Z) and the orientation Rot(u,v,w) in radians with respect to the robot base frame.

- **Pattern Detection:** The find_chessboard() function detects the chessboard pattern in the image, refines the corners and returns Rotation and translation matrix of the camera eye frame for each of the corresponding image using the previously computed intrinsic matrix.

- **Computing Hand Eye Matrix:**

  After the Camera and calibation object transformations and robot pose to base transformations are performed, these datas are fed to the OpenCV library calibrateHandEye() function as homogeneous matrix which returns the Rotation and Translation vectors for the camera frame with respect to the flange eye using the algorithm mentioned in Equation 2.19
  Reference code file: ***Hand_Eye_Calculation.py***
  The Hand-Eye matrix was found as :

$$Hand\_Eye\_matrix = \begin{bmatrix} 1.000 & -0.005 & -0.019 & -1.766 \\ 0.019 & 0.016 & 1.000 & 103.341 \\ -0.005 & -1.000 & 0.017 & 124.898 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.3}$$

4.4. Image Processing Workflow

To check the accuracy of the hand eye and camera calibration fed into the coordinate transformation calculations and developed image processing algorithms, a calibration check program was created to ensure its performance.

This program stands as the fundamental concept for moving ahead with the pick and place operations on the conveyor system integrated with the robot. The basic flow of the program is illustrated in the Fig. 4.9

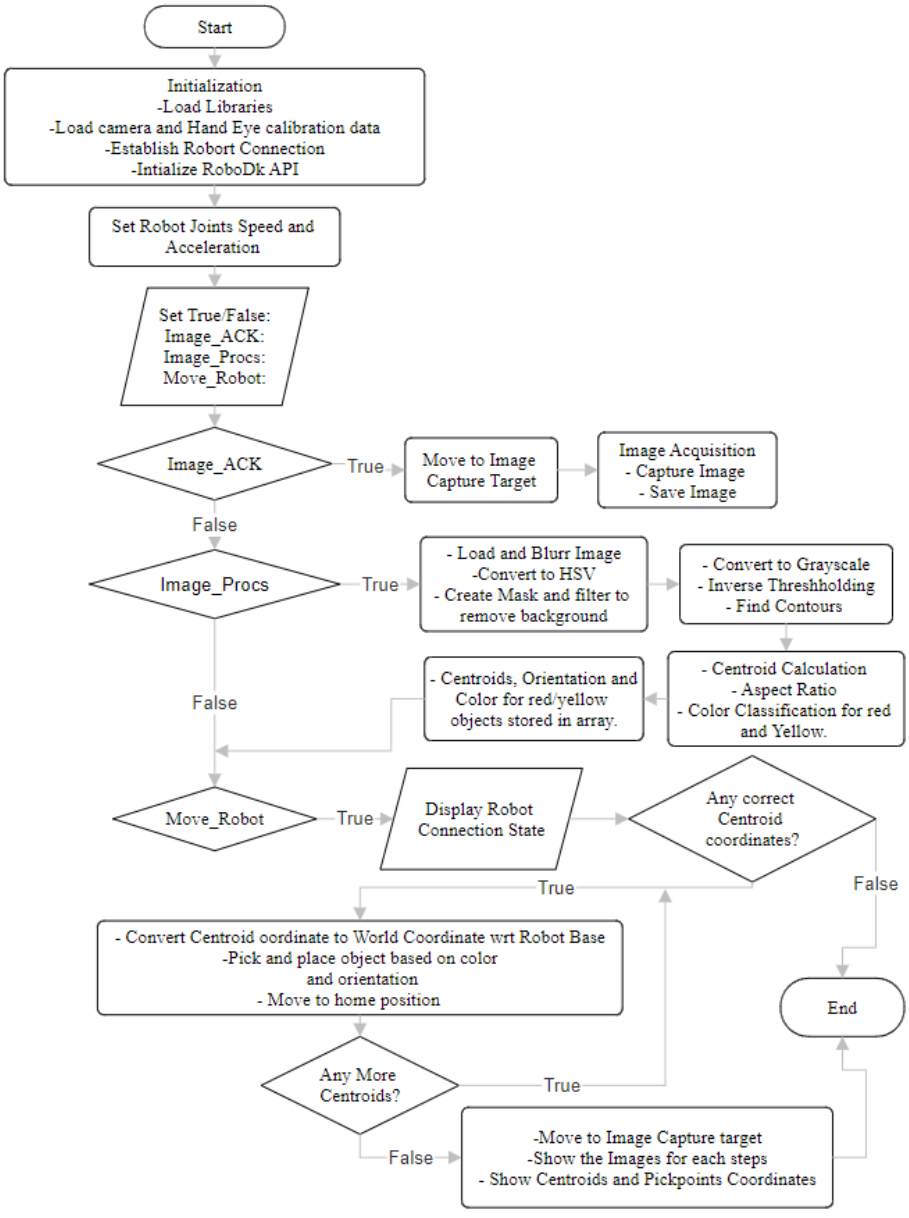


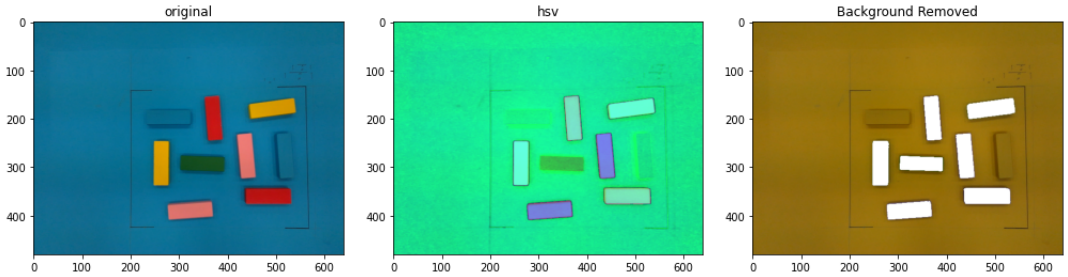
Figure 4.9. Flowchart for process of detecting and distinguishing blocks

When Image_ACK and Image_Procs are both set to True, the program initiates a detailed sequence for image acquisition and processing.

First, the robot moves to the predefined photo position, ensuring optimal conditions for capturing the image. The program then activates the camera and begins capturing frames, displaying them in a real-time stream. When the user presses the 's' key, the current frame is saved as an image file to a specified directory, and the camera stream is terminated.

Subsequently, the saved image undergoes processing to identify objects of interest. The image is converted to the HSV color space to facilitate color-based segmentation. A mask is applied to isolate the desired color range, and contours are detected from the masked image. The program calculates the centroids of these contours, determining the position and orientation of each object. It then classifies objects based on their color, assigning them to either the 'Red' or 'Yellow' category. This thorough image processing step ensures that the robot has accurate data on the location and orientation of objects, which is crucial for the subsequent pick-and-place operations ahead.

Below shown are the images during each step of processing to extract the object of interest and its centroid.



(a) *Original captured image* (b) *Converted to HSV Image* (c) *Extraction of objects profile*

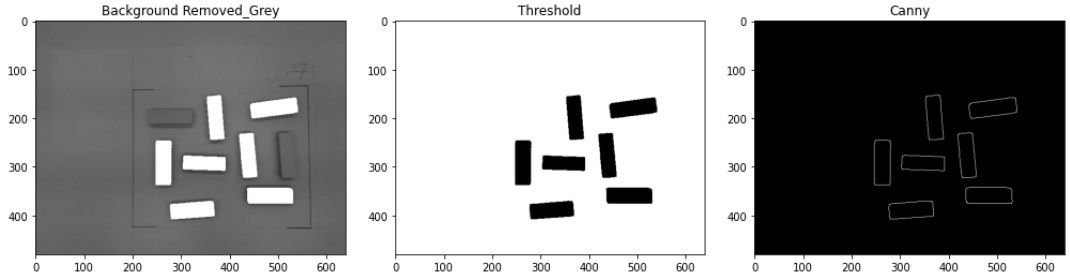
Figure 4.10. Image Processing Images Sequence 1

Here the images is taken on a blue background. To distinguish the objects from the background the color mask parameters are as below:

$$Lower\_blue = (32, 50, 80); \quad Upper\_blue = (140, 255, 255) \tag{4.4}$$

The Next Sequence shows the grayed out objects profile preparing it for threshold operation. A Inverse threshold was applied for a pixel value range of (200,240) such that the white pixels(255) only are inverted to black pixel values(0) and all other pixel values to white(255).

As the profile were distinct, the contours were found using Canny Edge Detection algorithm. The bounding boxes are drawn for the edges and mathematical calculations are performed to determine the aspect ratio for the purpose of orientation and Centroid Coordinates (x,y) (i.e. the pixel coordinates) for pick and place operation. . At the same time the area of the object is calculated and cropped to extract the average RGB value which helps to distinguish the red and yellow Object contour areas.These three values are passed to the target_set() function which is discussed in the next section in detail.



(a) *Objects profile grayed* (b) *Inverse Thresholded Image* (c) *Canny Edge Detection*

Figure 4.11. Image Processing Images Sequence 2

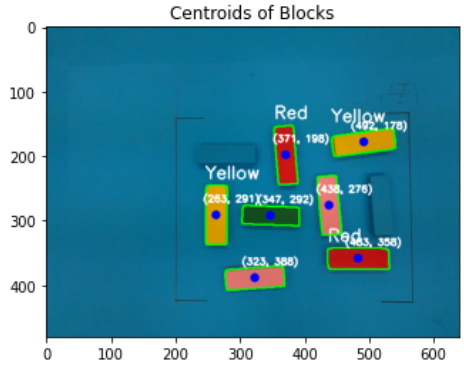


Figure 4.12. Drawn centroid coordinates; color information and object contours for objects of interest

The table 4.3 shows the detected objects corresponding target coordinates, the color and the orientation (H: Horizontal V: vertical) for Gripper rotation as of Fig. 4.12. The Objects not of Interest (NOI) are separated on the basis of the average RGB value.

Color Identification Criteria:

$$\begin{aligned} \textit{For Color Red: (RGB value)} &= (R \geq 100, G \leq 50, B \leq 50) \\ \textit{For Color Yellow: (RGB value)} &= (R \geq 100, G \geq 100, B \leq 50) \end{aligned} \tag{4.5}$$

S.N.	Centroids	Color	Orientation	Average RGB	Target Coordinates (wrt UR3e Base)
1	(483,358)	Red	H	(167,26,28)	(344.16,-250.90,0.64)
2	(438,276)	Light Pink	V	(182,107,108)	NOI
3	(492,178)	Yellow	H	(191,146,12)	(497.66,-259.35,2.95)
4	(371,198)	Red	V	(155,40,47)	(481.12,-155.93,4.65)
5	(347,292)	Dark Green	H	(18,77,40)	NOI
6	(323,388)	Light Pink	H	(188,117,119)	NOI
7	(263,291)	Yellow	V	(146,133,39)	(402.25,-63.30,5.13)

Table 4.3. Centroids and Targets for detected objects

4.5. Pick and Place Workflow

For this application, different coloured rectangular objects available at the lab were used. The main aim here was to pick only red and yellow coloured objects out of the all other and place them back into the rack.

Due to the limitations of the gripper movement, some constraints were defined in the beginning and solutions were formulated to counter them which are discussed in section 5.2.

The target_set function is responsible for converting the centroid coordinates of an object in an image to the corresponding coordinates in the robot's coordinate system, and then moving the robot to pick up and place the object.

4.5.1. Pickpoint Calculation

Function Definition and parameters:

- **u, v:** These are the pixel coordinates of the centroid of the object in the image.
- **color:** This parameter specifies the color of the object, which determines where the object should be placed (either at the ReleaseRed or ReleaseYellow position).
- **orient:** This parameter indicates the orientation of the object (either 'H' for horizontal or 'V' for vertical), which affects how the gripper should be oriented to pick up the object.

Convert Image Coordinates to Camera Coordinates

- **normalized_coords** is obtained by multiplying the inverse of the camera matrix K (Equation 4.2) with the pixel coordinates [u, v, 1].
- **x_c, y_c, z_c** are the camera coordinates obtained by scaling the normalized_coords with the predefined depth value.

  Depth Value to pick objects from Table Surface: 512mm
  Depth Value to pick objects from Conveyor Belt: 190mm

Depth Value to pick Bolt-deficient Blocks objects from Conveyor Belt: 165mm

- **cam_coords** is the homogeneous coordinate representation of the camera coordinates.

Convert Camera Coordinates to World Coordinates

- **world_coords** is obtained by multiplying the transformation matrix H_flange_to_eye (i.e Hand Eye Matrix; Equation. 4.3)with the cam_coords. This converts the camera coordinates to world coordinates, relative to the robot's end-effector (flange).

Convert World Coordinates to Robot Base Coordinates

- **target_coords** is obtained by multiplying the transformation matrix H_base_flange(pose of the robot during image capture; Equation 4.6) with the world_coords. This converts the world coordinates to the robot's base coordinates.

$$Hand\_base\_flange = \begin{bmatrix} 0.000 & 0.000 & 1.000 & 305.299 \\ -1.000 & 0.000 & 0.000 & -131.050 \\ -0.000 & -1.000 & 0.000 & 620 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.6}$$

- **target_pos** is the (X, Y, Z) position of the target in the robot's base coordinate system.

4.5.2. Robot Motion

Subsequent delays in the range of 0.25 - 1 seconds are added in between the steps below to carry out smooth robot movement flow along with speed variations to pick and place the objects securely.

- **Step 1:** Set the Target Pose for the Robot
  - **target** is the robot target item 'PickPoint'.
  - **pose_ref** is the current pose of the robot.
  - **pose_ref.setPos(target_pos)** sets the position of the target to the calculated target position.
  - **target.setPose(pose_ref)** and **target.setAsCartesianTarget()** update the pose and set it as a Cartesian target.
- **Step 2:** Move the Robot TCP to Home Position
- **Step 3:** Orient the Gripper Based on object orientation

- Depending on the orient parameter, the gripper is oriented either horizontally or vertically using predefined programs **turnV_prog** and **turnH_prog**.

- **Step 4:** Open the Gripper
  - **robot_item.setDO(10, 1)** sends a signal to open the gripper.
- **Step 5:** Move the Robot TCP to the Target Position
- **Step 6:** Close the Gripper to Pick Up the Object
  - **robot_item.setDO(10, 0)** sends a signal to close the gripper, picking up the object.
- **Step 7:** Move the Robot TCP Back to the Home Position
- **Step 8:** Orient the Gripper Back to Horizontal
- **Step 9:** Move to Release Position Based on Object Color/Size/ Deficient bolts
  - Depending on the **color** parameter, the robot moves to the release position for either the red or yellow object.
- **Step 10:** Open the Gripper to Release the Object
- **Step 11:** Reorient the Gripper to Horizontal
- **Step 12:** Move back to Home Position

4.6. Pick and Place from Conveyor

The previous experiment was conducted on static images to check and the accuracy of pick and place operation and optimize the image processing algorithm. Moving ahead, to replicate a simple industrial scenario .i.e a dynamic environment with objects moving on a conveyor the setup had to be modified.

The figure 4.13 shows the simulation setup of the robot gripper-camera assembly with the conveyor designed. Here, The Image capture position moved forward and shifted down to capture the conveyor image area accurately.

In this station tree, few additional programs and target points were added which are listed and explained below:

1. **Image Stream:** Moves the robot TCP to Video Stream position.
   [400.300000, -131.050000, 250.000000, -69.282000, 69.282000, -69.282000]
2. **ReleaseSize:** This target point defines the rejection point of the under-size object dimensions detected.
   [320.000000, -70.000000, 150.000000, -69.281000, 69.285000, -69.284000]

3. **ReleaseBolt:** This target point defines the rejection point of the deficient bolts on the detected object.
   [250.000000, -180.000000, 60.000000, -69.281000, 69.285000, -69.284000]

Additional movement programs:

1. **Convey_ON:** This program links the robot with the conveyor through a relay. The actuation signal to the relay is provided through the Digital Output (DO) pin 0.
   This provides the 24V output when set as True.
2. **Convey_OFF:** This programs sets the DO 0 pin to False which de-energises the relay and stops the conveyor.

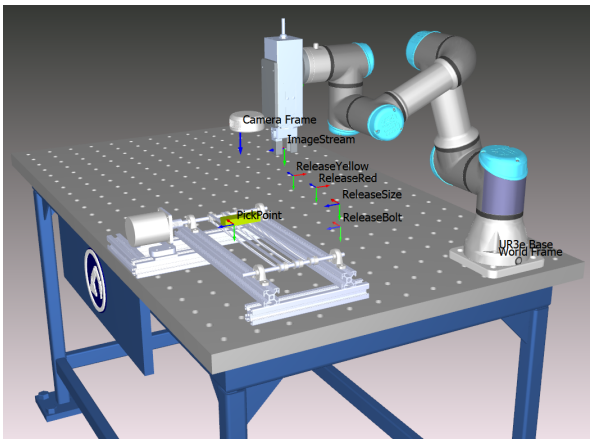


Figure 4.13. Simulation setup of Robot integrated with conveyor

Three different experiments were conducted with the robot conveyor assembly which shall be discussed in the next section.

5. Results and Discussion

5.1. Object Detection and Placement Scenarios on Conveyor

Here, three different scenarios are replicated for object detection and pick and place from a moving conveyor. The setup, program- flow and the results will be discussed in this section. The actual setup of the robot-gripper-camera assembly with the conveyor is shown in Figure 5.1.

The algorithm for all these different scenarios are integrated into a single python program which links the corresponding robodk station tree.
The Configuration setup for the program is:

Video_ACK_PRCS = True
Conv_Move = True
Robot_Move = True

The above programs must always be set to True and the below program branches are set to True/False by the user as per the desired task to be executed which must be defined as True and others as False.

Program_RedYellow = True/False
Program_Object_Size = True/False
Program_Missing_Bolt = True/False

Reference code file: ***Main_Program_Conveyor.py***

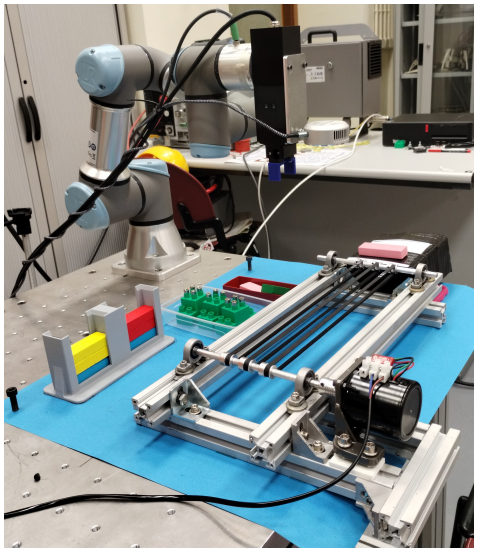


Figure 5.1. Real Setup of Integration of Conveyor with Robot

5.2. Detection based on Red and Yellow Objects

In this scenario, the algorithm is developed to detect and pick-place the red and yellow objects only from the conveyor.

The user defined configuration setup for the program is:

Program_RedYellow = True
Program_Object_Size = False
Program_Missing_Bolt = False

The TCP is moved to the image stream target coordinate and images are streamed at 30 fps. A detection window is defined to focus the required conveyor belt area. This is done by cropping the original stream.
The cropping parameters are given below:

- **Original Image Size:** (640 X 480) px
- **Cropped Image Coordinates:**
  Top left corner (x1,y1) = (180,50) px
  Bottom right corner (x2,y2) = (470,430) px
- **Cropped Image Size (Detection Window):** (280 X 380) px

The figure 5.2 shows the original image stream and the detection window for image processing ahead.

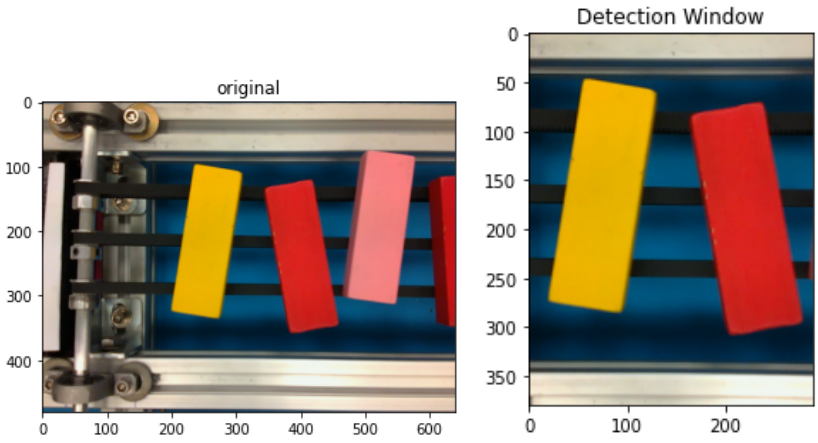


Figure 5.2. Original and Cropped Image

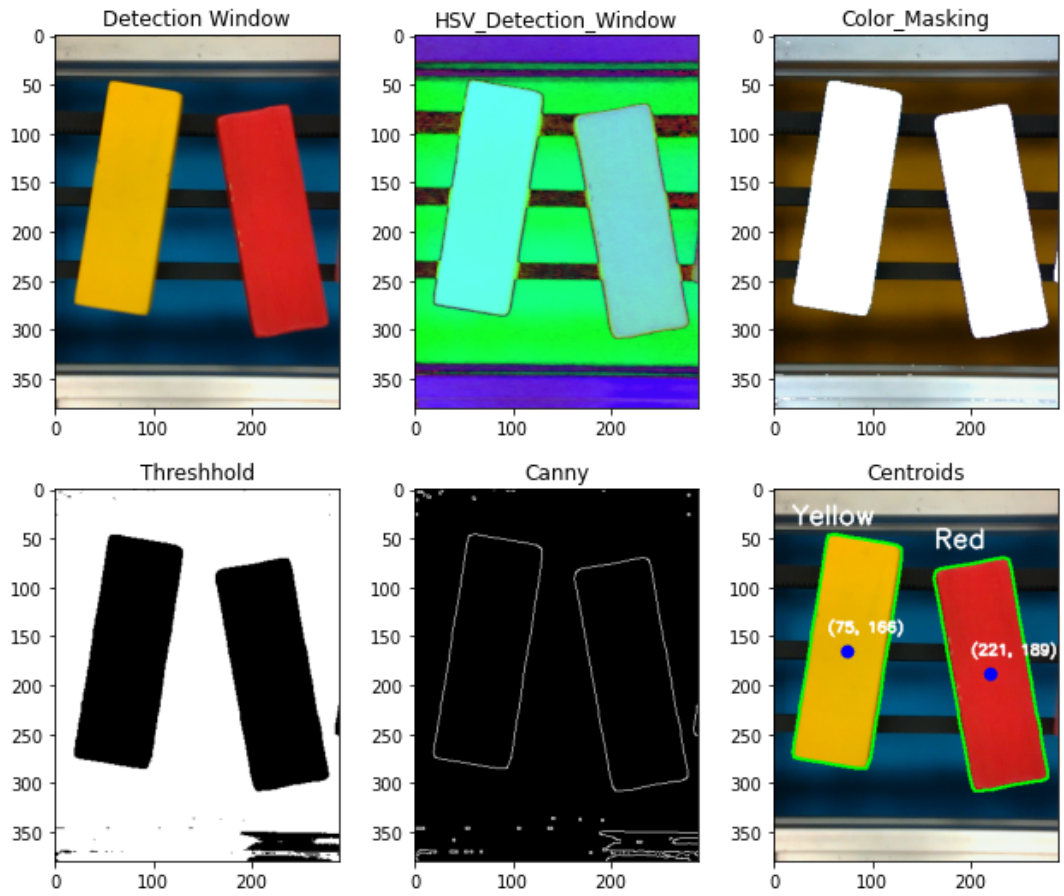


Figure 5.3. Image processing steps for Red and yellow colored objects detection from conveyor

The Steps are explained briefly in the below points:

1. Image frames are captured from Video Stream robot target cordinates.
2. Image Cropped to extract the detection window.
3. Conversion of BGR Image to HSV
4. Color masking operation to extract the foreground object from background. Eqtn. 4.4
5. Canny Edge Detection
6. Aspect ratio and Centroid Calculation
7. Region of Interest Extraction to calculate the average RGB color values
8. Classification of Objects based on Color Criteria. Eqtn.4.5
9. Draw the bounding box and centroid coordinates on the Detection window.
10. Pass the centroid, color and orientation to target_set() function to perform robot pick and place operation.

The program picks up only the red and yellow objects from the conveyor and allows other colored objects to pass through. The Fig. 5.4 shows how the other colored objects and red-yellow are visualized in the detection window.

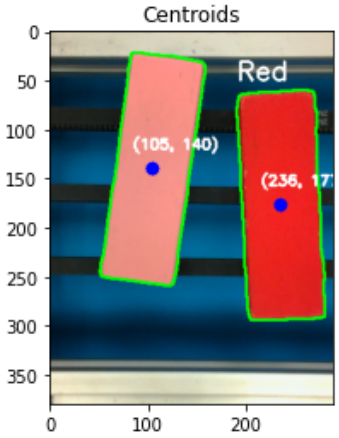


Figure 5.4. Detection Window computations for red and other colored object

5.3. Detection based on Object Size

In this scenario the objects less than the standard sizes are rejected from the running conveyor line. The configuration setup for the program is as:

Program_RedYellow = False
Program_Object_Size = True
Program_Missing_Bolt = False

The steps for the Image processing is same as the section 5.2. However, for the object size detection an additional criteria is defined. The contours areas are calculated and the criteria is defined.

Standard Contour area of objects: ≈ 18000
Rejection criteria for Contour area: ≤ 16000

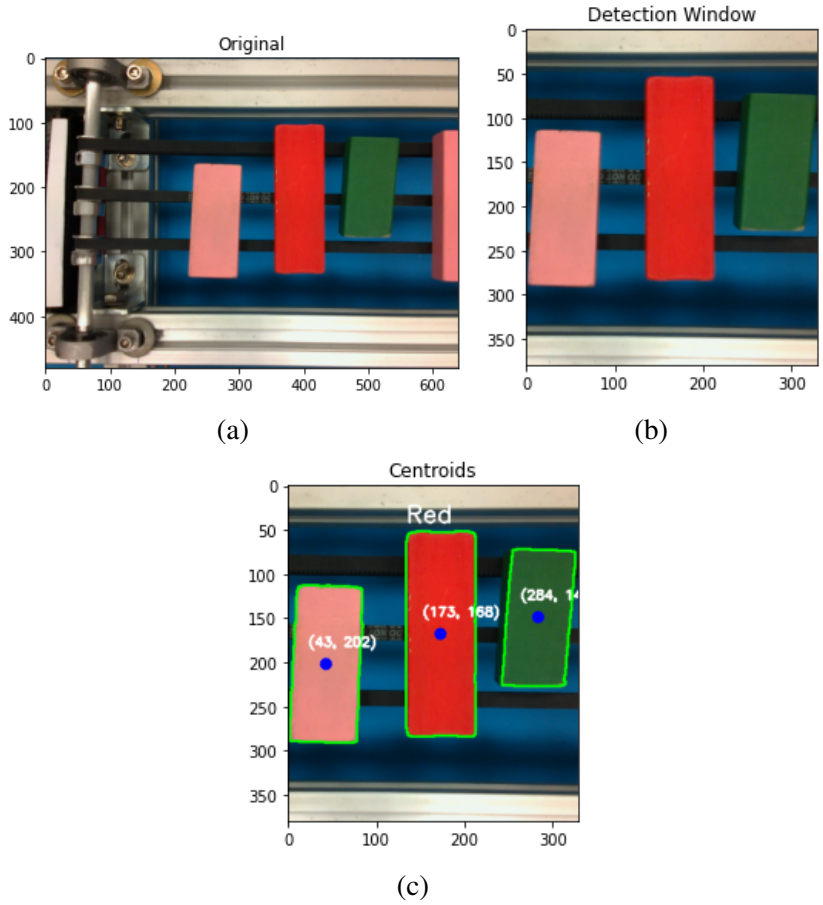


Figure 5.5. a) Original Image b) Cropped Image c) Centroid and Object Detection based on object size

Here, the objects with centroid coordinate (43,202) with contour area: 13357.5 and (284,148) with contour area: 11045.0 are rejected from the conveyor.

5.4. Detection based on Deficient bolts in the object

In this scenario, a algorithm is developed to detect the object and the bolts fitted to the object. The standard object contains four bolts, in case of deficient bolts or no bolts found in the object the robot shall reject found defective object from the conveyor line.

For the purpose of replication, steel bolts are installed into the green-colored objects and placed on the conveyor.

The Configuration setup for the program is:
Program_RedYellow = False
Program_Object_Size = False
Program_Missing_Bolt = True

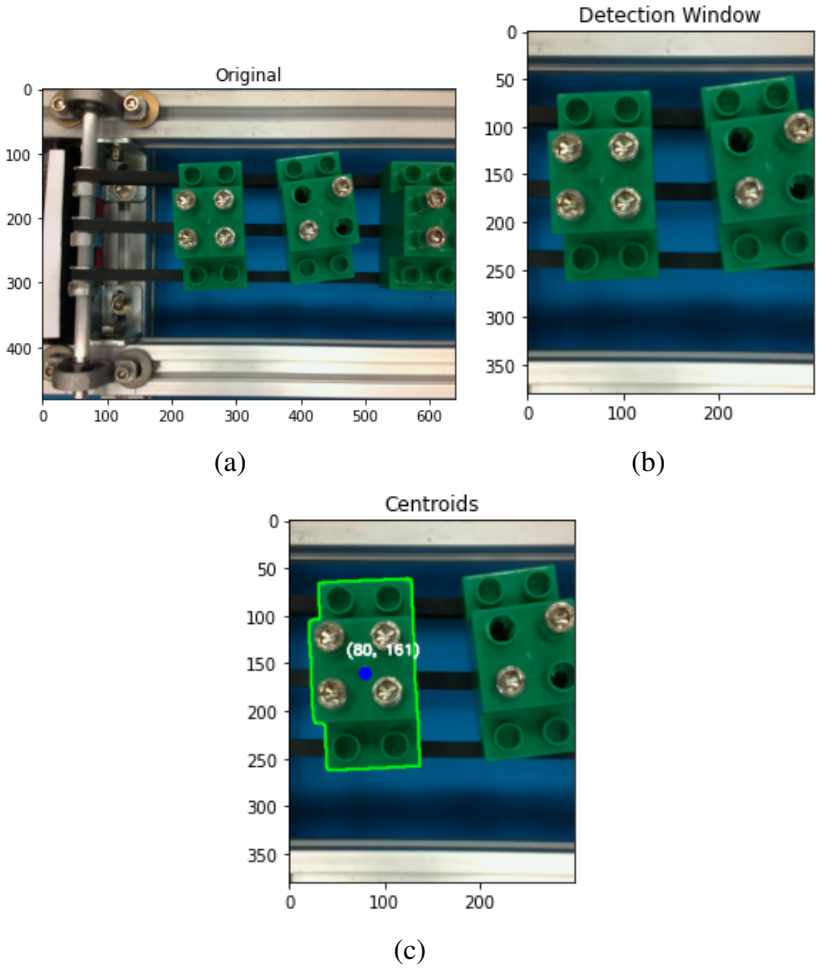


Figure 5.6. a) Original Image b) Detection Window c) Object Detection and Centroid Calulation

The centroid is calculated using the same image processing steps as mentioned in section 5.2. After the object has been detected and the centroid value stored, the next step is to detect the bolts on the object using additional Image processing operation. This is performed by extracting the pixel coordinates of the object as Region of Interest (ROI), widening the periphery and passing it to the function **search_bolt()** function.

The Fig. 5.6 shows the extraction of object profile, draws a bounding line for the object and calculates the centroid coordinates.

The basic steps involved in the search_bolt function is briefly explained below:

1. Retrieve the Object widened ROI

2. Convert the ROI image to HSV

3. Apply Color mask to retrieve only the bolts. The color mask HSV range parameters are given below:

$$Lower\_silver = (32, 50, 80); Upper\_silver = (140, 255, 255) \tag{5.1}$$

4. Apply histogram equalization.
   This process helps to increase the contrast and visibility of the image by redistributing the pixel values.

5. Median blurring the image.
   Since the Video streams at processed at 30fps, it creates fluctuating edge pixel values. To preserve the edges well and remove noise from the image, Median filter is utilized.

6. Threshold the blurred image. Inverse Binary Threshold is used in the range of (0,100) for preparation for edge detection.

7. Applying Gaussian Blur . Kernel size: (9,9)

8. Canny Edge Detection

9. Definition of Bolt Contour Detection Criteria.
   Contour area in the range if (500,1000) are chosen.

10. Bolt Circle Contour are approximated
    Centres recorded and circles are drawn for clear and accurate detection. The circles with radius in the range of (12,20) are approximated and drawn over the image.

11. The objects with deficient bolts are classified based on numbers of circles approximated. In this case, less than four.

The Fig.5.7 shows the image processing steps in order to extract information on the the number of bolts present on the object.

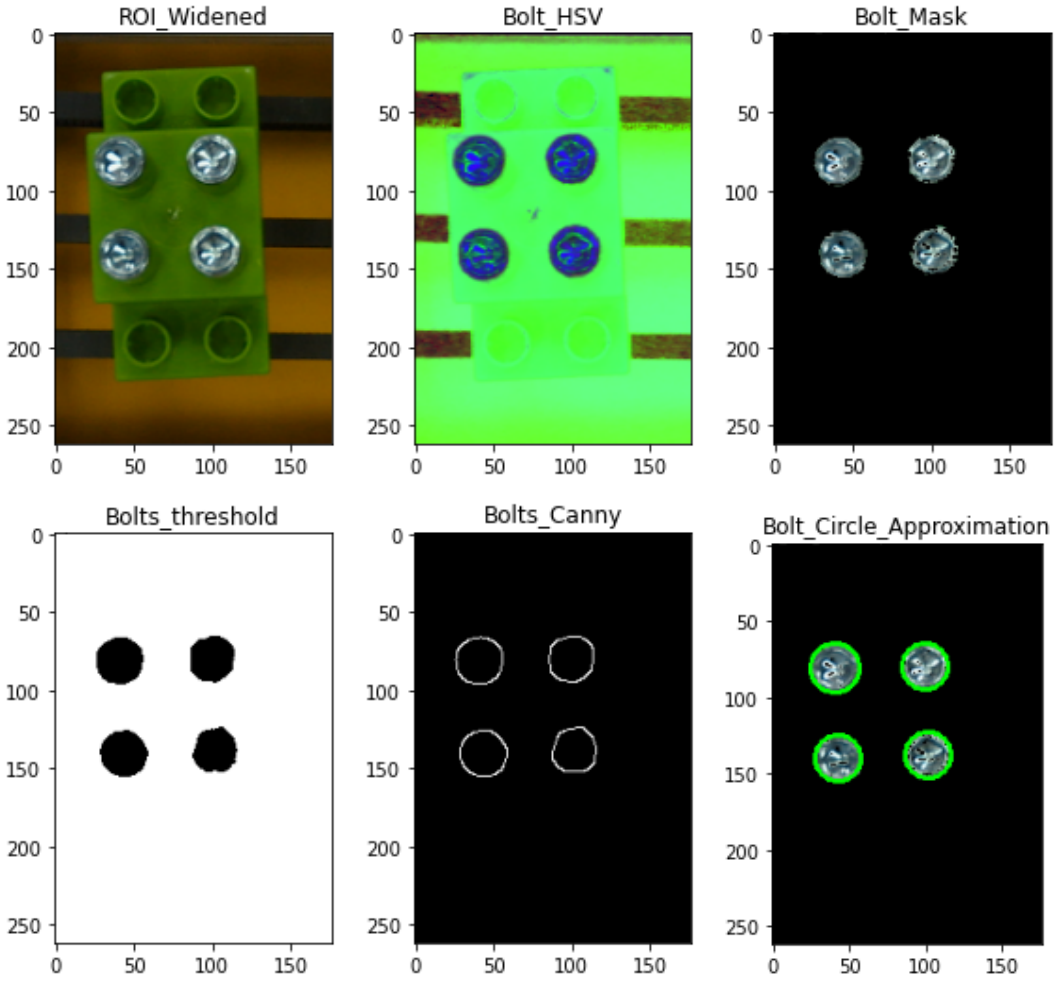
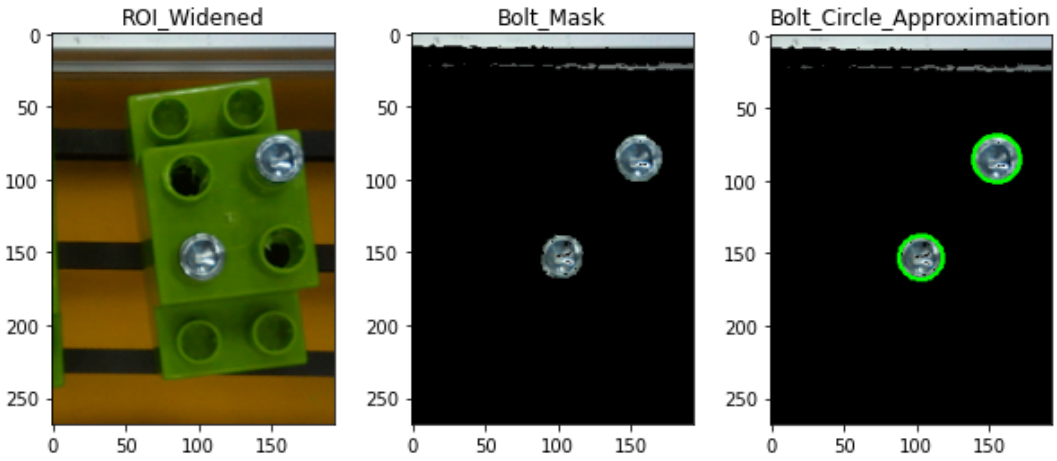


Figure 5.7. Image processing steps for deficient bolts in objects on the conveyor



(a) (b) (c)

Figure 5.8. a) ROI b) Color Masked c) Bolt Circles Approximation

The figure 5.8 depicts the defective object with less bolts than the standard ones and is rejected from the line.

5.5. Discussions

5.5.1. Detection of Rectangular Blocks and Centroid Calculations

Actual centroid and Detected Centroid Comparision Table:

The high accuracy in detecting the rectangular blocks and calculating centroids demonstrates the robustness of the image processing techniques used. This ensures reliable pick-and-place operations by the robotic arm.

5.5.1.1. Different Rectangular Objects

Since the tests were carried out with Red, Yellow, Pink and Green objects, the algorithm on different size and colored objects was tested to analyze the detection and centroid calculations. In the Fig. 5.9, we see the objects edges were detected accurately and centroids were placed accurately in case of robot pick place operation.

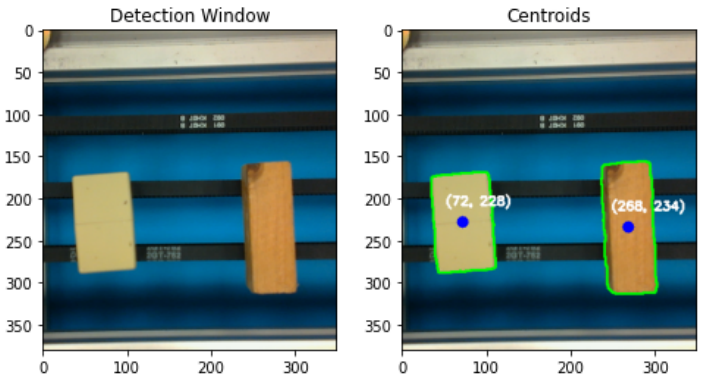


Figure 5.9. Detection Window and Centroids for different rectangular blocks

5.5.1.2. Effect of Detection Window Increase/Decrease and Object Spacings

The Detection window size impacts the accuracy of pick and place action as the calculation of centroid point is affected. The objects placed far from the vertical axis of the camera eye accounts for additional area due to the object height. This effects the edges of the object to have minor inaccuracies which in result provides shifted centroid coordinates. The effect can be shown in below Fig. 5.10 where the distant yellow object has a slight shift in the centroid point.

As in case of objects touching each other, both the objects are not detected as the edges of the object coincide with each other and do not fall under the defined criteria of contour detection. The object spacing should be such that the gripper jaws in fully open position should be able to grasp the object without touching other objects nearby.

The minimum distance between the objects as per the fully opened Jaw was calculated as (placed vertically on the conveyor):

Object Width (w): 25mm
Jaw span (s) : 38mm
Clearance (c): 6 mm (accounting for minor inaccuracies and safe operation)
Minimum Object Spacing: (s/2)-(w/2)+ c = 12.5 mm

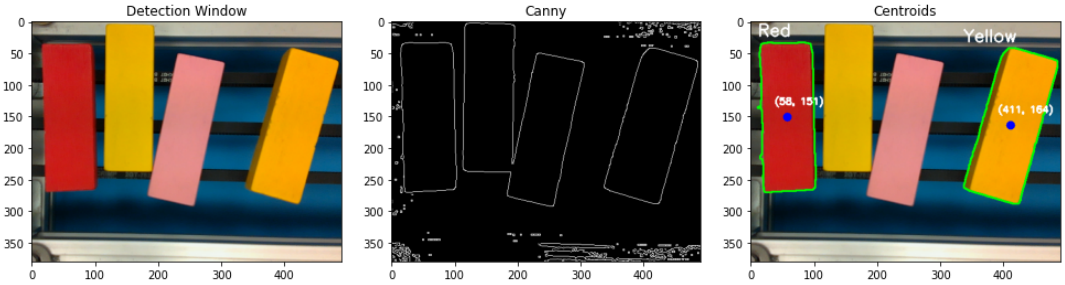


Figure 5.10. Objects with increased Detection Window and Closely Spaced or Touching

5.5.2. Detection of coloured objects similar to the background

The objects which have a similar color to the background are difficult to detect and pose difficulties during color masking. In our case a blue background was used, and coincidentally the available blue objects had a similar blue color range as of the background. Attempts to design a color mask differentiating the similar color range posed a challenge as other color ranges of requirement fall out of range in process. Fig. 5.11 shows that the objects similar to the background are not detected in the color mask, hence no computations are performed.

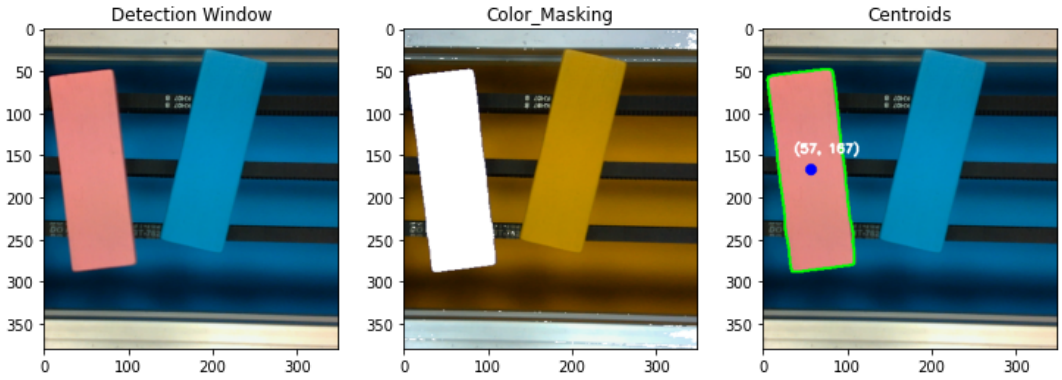


Figure 5.11. Objects with similar Background color

Normally Blue and Green backgrounds are used for digital image processing. The choice for using a blue background was made due to the below advantageous points:

- **Less reflective:** They perform better where lighting is difficult to control and reflections need to be minimized.
- **Low light situations:** They perform better in low-light situations as they are less bright and require less light.

5.5.3. Rejection Based on Deficient Bolts

As the color mask was used to extract the bolts contour, the performance was accurate in static imaging. However in a video stream process the mask caused flickering effect and unstable edges which triggered false detections of the blocks. Fig. 5.12 shows the false positive detection instance for a block in ROI.

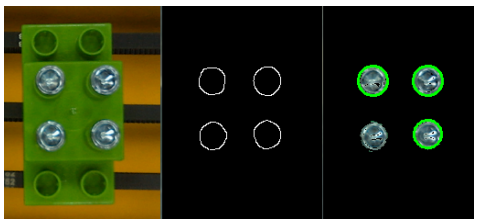


Figure 5.12. False positive instance during deficient bolts detection

To reduce the flickers and noise before the edge detection some image processing techniques were executed which improved the detection accuracy.Histogram equalization combined with Median Blurring reduced the noise and jitter in the image stream. Next circles are approximated and the number of centres were used as basis of measure rather than directly using the raw canny edges result.

5.6. Challenges Faced

Constraint 1: Gripper Jaw Span

The objects to be picked up had dimensions (L x W x H) of 75 x 25 x 15 mm. The original gripper jaw span was 14 mm, allowing the object to fit only along its height. Despite the low tolerance of 0.5 mm, the algorithm was executed successfully, and the objects were grasped with minor inaccuracies. This demonstrated that the transformation from pixel coordinates to robot base coordinates was accurate, and that increasing the tolerance distance could eliminate these minor positional target errors. To address this issue, the gripper jaw span was modified from 14 mm to 38 mm ensuring a firm grip of the object when the gripper is in fully closed position. The below figure 5.13 shows the modification of the gripper jaws.

Previously: Gripper Fully Open: 14mm, Gripper Fully Closed: 2mm

Modified: Gripper Fully Open: 38mm, Gripper Fully Closed: 22mm

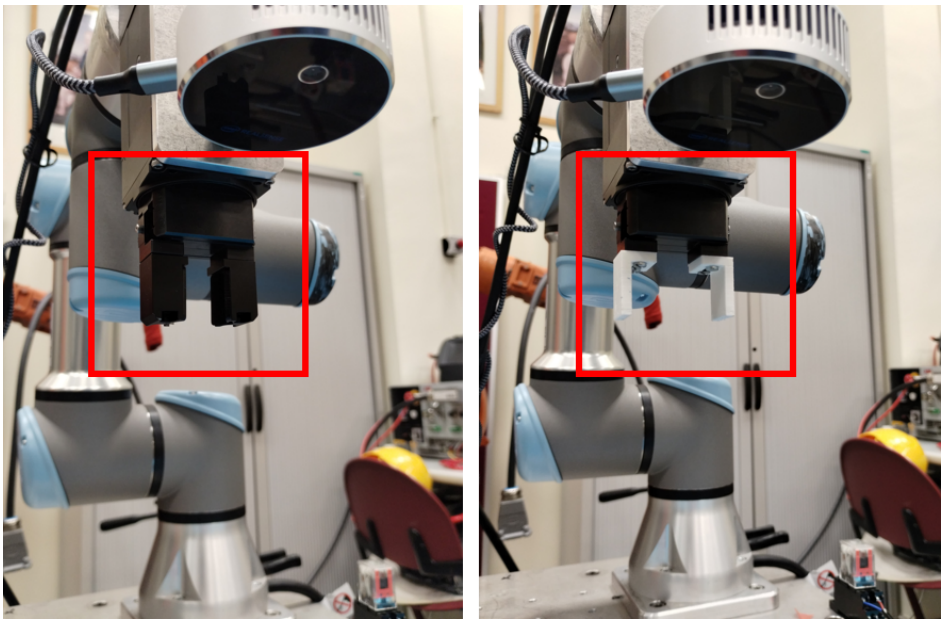


Figure 5.13. Gripper Jaw Modification

Constraint 2: Gripper Rotation angle

The next constraint encountered was the rotational movement of the gripper, which was programmed to rotate only 90 degrees horizontally and vertically. This limitation meant that objects could only be picked up and placed in strictly horizontal or vertical positions. Efforts to reprogram the gripper using analog inputs were unsuccessful due to software issues. However, with the increase in jaw span, the objects could now be positioned at an angle of ±18 degrees. This adjustment allowed for slight deviations from the horizontal and vertical axis angles.

Constraint 3: Camera Depth Range Limitation

The LIDAR depth camera could measure depth from a minimum distance of 50 cm. Consequently, the image capture point was positioned as high as possible. However, due to the limited reach of the robot arm, the depth data could not be extracted effectively. Since the objects were placed on a table surface or a conveyor belt, the depth remained constant for all objects. Therefore, the depth was manually measured and incorporated into the transformation calculations.

6. Conclusion and Future Work

6.1. Conclusion

In this thesis, a comprehensive approach to enhancing the functionality and precision of a vision-guided robotic arm was developed and validated through a series of tasks. The key accomplishments of this research are summarized as follows:

- Successful implementation of advanced image processing techniques to accurately detect blocks and perform centroid calculation. This enabled precise pick-and-place operations, significantly reducing error margins and increasing operational efficiency. The accuracy of the centroid calculations proved the robustness of the approach, ensuring reliable object handling by the robotic arm.
- The integration of color detection algorithms allowed the robotic system to accurately identify and reject blocks based on their color, specifically red and yellow. This task was achieved with high precision, demonstrating the system's capability to perform real-time sorting based on visual characteristics.
- By employing contour area calculations, a reliable method for rejecting blocks based on size criteria was established. This approach effectively filtered out blocks that did not meet the specified dimensions, showcasing the utility of contour analysis in size-based sorting tasks.
- The detection of deficient bolts on blocks was achieved through in depth image analysis, ensuring that blocks with missing bolts were correctly identified and rejected. This task is vital for quality assurance in manufacturing, as it ensures that only properly assembled products proceed down the conveyor line.

6.2. Future Directions

- Further work could focus on enhancing the robustness of color detection algorithm under varying conditions, improving contour analysis for irregular shaped objects and refining bolt detection methods to reduce false positives and negatives.
- Machine learning has the potential to significantly enhance the adaptability and performance of the robotic system. Integrating machine learning models along with classical image processing techniques allows the system to perform complex detections. These models could adapt to new sorting criteria over time and improve the system flexibilty along with recognizing wider variety of objects.
- The possibilty of implementing Eye to Hand setup (i.e fixed camera eye) along with integration of different gripper design (soft gripper, vacuum or pneumatic) to explore the pick-place operation with wider variety of objects.
- Modifying the belt of the conveyor prototype to reduce the image jitters over the detection window. Additionally a encoder could be integrated with the conveyor to perform object detection along with tracking of the object on the conveyor for a efficient inspection.
- Using higher resolution cameras to capture detailed images (or even 3D reconstruction) could be used for advanced detection algorithms such as deep learning based object detection to ensure reliability of the inspection process.

BIBLIOGRAPHY

[1] Zivid, “"See anything and everything with robot-mounted vision".” https://www.zivid.com/robot-mounted-3d-vision, 2024. Accessed: 15-06-2024.

[2] Q. Magazine, “"How AI and Machine Vision Impact Vision Robotics".” https://www.qualitymag.com/articles/96664-how-ai-and-machine-vision-impact-vision-robotics, 2024. Accessed: 2024-07-02.

[3] K. S. DELTA, “"From 3D Model to 2D Image".” https://keasigmadelta.com/blog/warp3d-nova-3d-at-last-part-1/, 2016. Accessed: 2024-06-10.

[4] S. I. SA, “"Automation Engineering and IT".” http://www.ipacv.ro/proiecte/robotstudio/textbooks/file/robot_motion.html, 2021. Accessed: 2024-06-13.

[5] X. Wang, G. Liu, Y. Feng, W. Li, J. Niu, and Z. Gan, “Measurement method of human lower limb joint range of motion through human-machine interaction based on machine vision,” *Frontiers in Neurorobotics*, vol. 15, 2021.

[6] Mathworks, “"Pinhole Camera Model".” https://www.mathworks.com/help/vision/ug/camera-calibration.html, 2024. Accessed: 2024-06-12.

[7] A. Segun, O. I. Sunday, E. O. Ogunti, and F. K. Akingbade, “Solution to bird pest on cultivated grain farm: A vision controlled quadcopter system approach,” *International Journal of Engineering & Technology*, vol. 7, no. 10, 2018.

[8] A. Yufka and O. Dobrucalı, *Indoor Localization System for Mobile Robots*. PhD thesis, 06 2008.

[9] A. Kumar, “"An Overview of Visual Servoing for Robot Manipulators".” https://control.com/technical-articles/an-overview-of-visual-servoing-for-robot-manipulators/, 2021. Accessed: 2024-06-16.

[10] R. Horaud and F. Dornaika, “Hand-eye calibration,” *The International Journal of Robotics Research*, vol. 14, p. 195–210, June 1995.

[11] I. Enebuse, M. Foo, B. S. K. K. Ibrahim, H. Ahmed, F. Supmak, and O. S. Eyobu, “A comparative review of hand-eye calibration techniques for vision guided robots,” *IEEE Access*, vol. 9, pp. 113143–113155, 2021.

[12] X. Lu, “A review of solutions for perspective-n-point problem in camera pose estimation,” *Journal of Physics: Conference Series*, vol. 1087, p. 052009, 09 2018.

[13] B. Zhou, Z. Chen, and Q. Liu, “An efficient solution to the perspective-n-point problem for camera with unknown focal length,” *IEEE Access*, vol. PP, pp. 1–1, 09 2020.

[14] S. Yuan, W. Zhao, J. Deng, S. Xia, and X. Li, “Quantum image edge detection based on laplacianof gaussian operator,” 12 2023.

[15] J.-D. Chang, S.-S. Yu, H.-H. Chen, and C.-S. Tsai, “Hsv-based color texture image classification using wavelet transform and motif patterns,” *Journal of Computers*, vol. 20, 01 2010.

[16] M. M. Medium, “"Color Image Segmentation — Image Processing".” https://mattmaulion.medium.com/color-image-segmentation-image-processing-4a04eca25c0, 2021. Accessed: 2024-06-16.

[17] U. Robots, “"Universal Robots e-Series User Manual Version 5.0.2.".” https://s3-eu-west-1.amazonaws.com/ur-support-site/41166/UR3e_User_Manual_en_Global.pdf, 2023. Accessed: 2024-06-20.

[18] DH-Robotics, “"RGI-14 User Manual".” https://en.dh-robotics.com/service/download, 2024.

[19] I. RealSense, “"Intel RealSense LiDAR Camera L515".” https://www.intelrealsense.com/lidar-camera-l515/, 2024.

[20] S. Electric, “"Industrial Relays".” https://www.se.com/uk/en/product-subcategory/2810-industrial-relays/, 2024.

[21] RoboDK, “"Simulate Robot Applications".” https://robodk.com/simulation, 2024.

[22] R. API, “"RoboDK API for Python".” https://robodk.com/doc/en/PythonAPI/index.html, 2024.

[23] Anaconda, “"The Scientific Python Development Environment".” https://anaconda.org/anaconda/spyder, 2024.

[24] P. Corke, “"Robotics Toolbox".” https://petercorke.com/toolboxes/robotics-toolbox/, 2020.

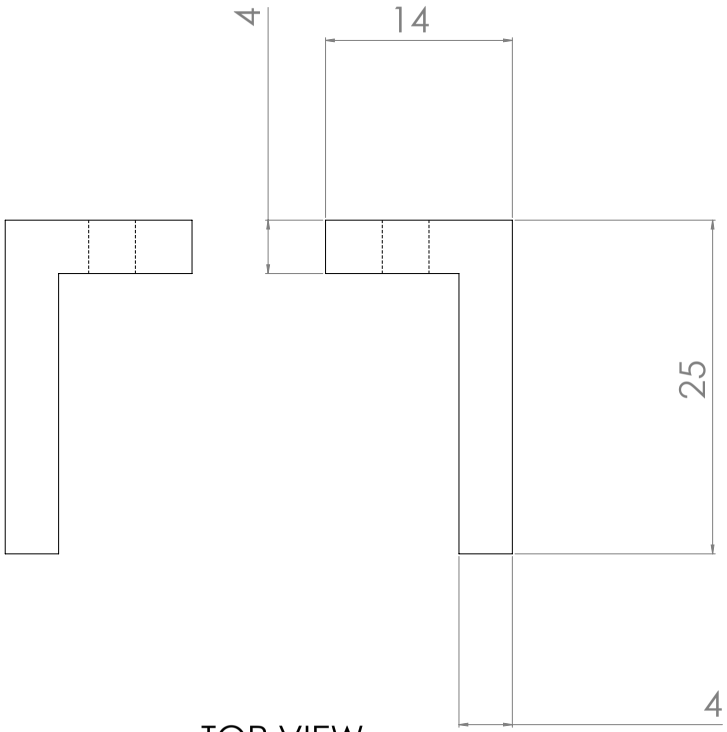
[25] RoboDK, “"Universal Robots".” https://robodk.com/doc/en/Robots-Universal-Robots.html, 2024.

[26] M. H. Jones, “"Calibration Checkerboard Collection".” https://markhedleyjones.com/projects/calibration-checkerboard-collection, 2018.

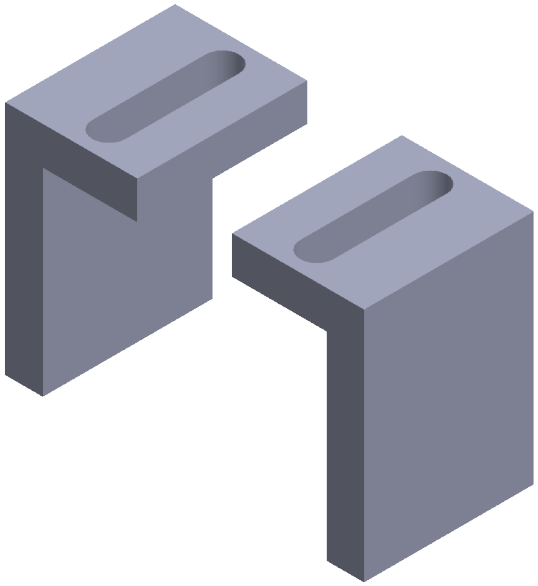
[27] Mathworks, “"Evaluating the Accuracy of Single Camera Calibration".” https://www.mathworks.com/help/vision/ug/evaluating-the-accuracy-of-single-camera-calibration.html, 2024.

7. Appendices

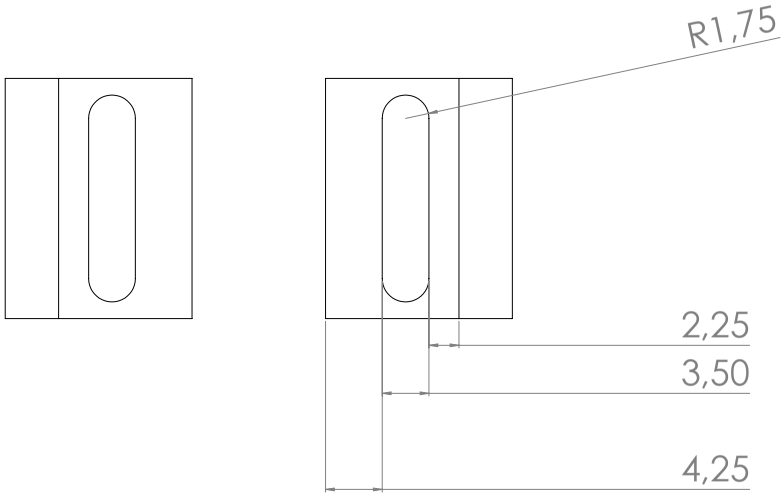
7.1. CAD Files



TOP VIEW



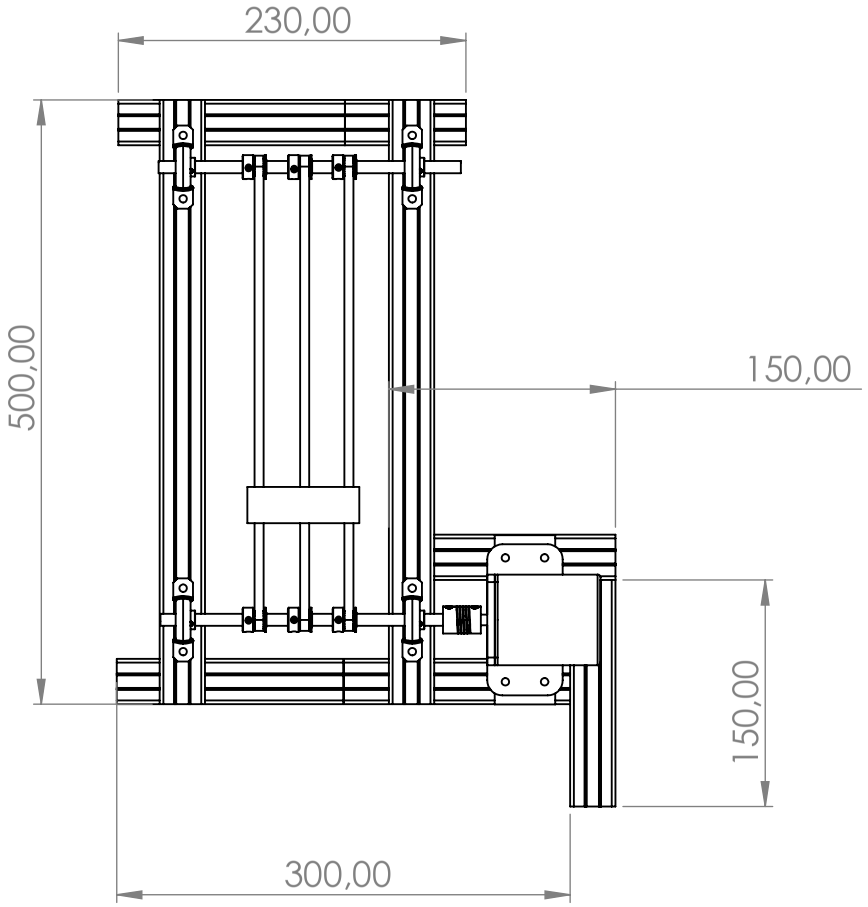
ISOMETRIC VIEW



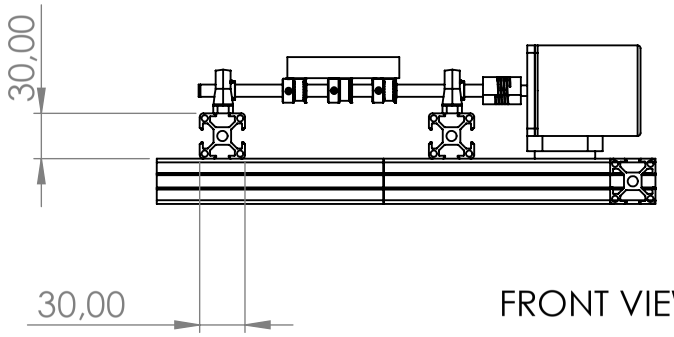
FRONT VIEW

All dimensions are in mm

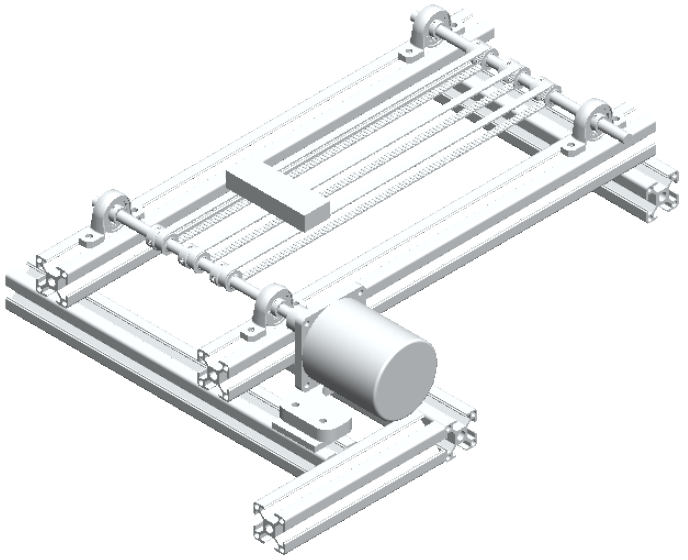
BIBEK GUPTA	Universidad de Oviedo	
EU4M	Gripper Jaw	
Master in Mechatronic Engineering		APRVD BY:
SCALE: 1:1		CHKD BY:
DATE: 11/07/2024		SHEET NO: 1 OF 1



TOP VIEW



FRONT VIEW

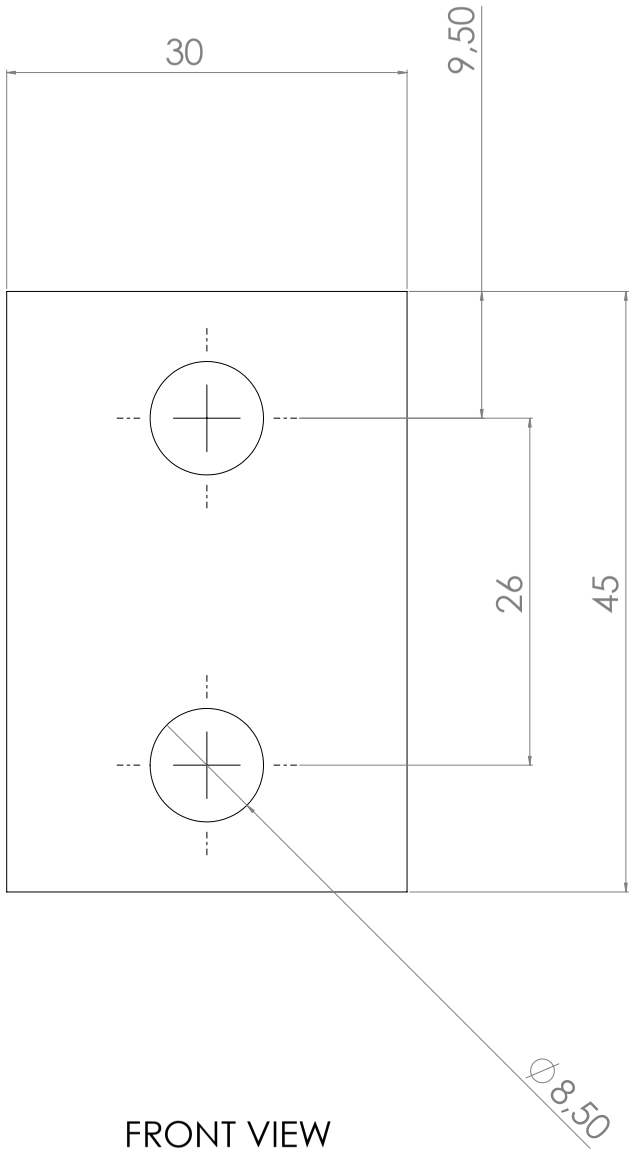


ISOMETRIC VIEW

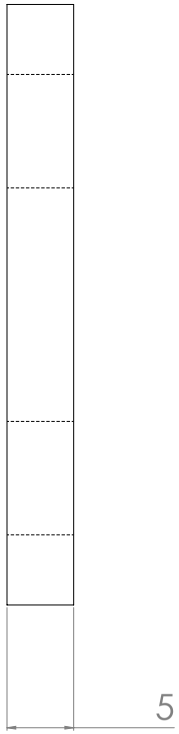
S.N	Aluminum Profile (30X30) Length	Quantity
1	500	2Nos.
2	300	1No.
3	230	1No.
4	150	2Nos.

All Dimensions are in mm

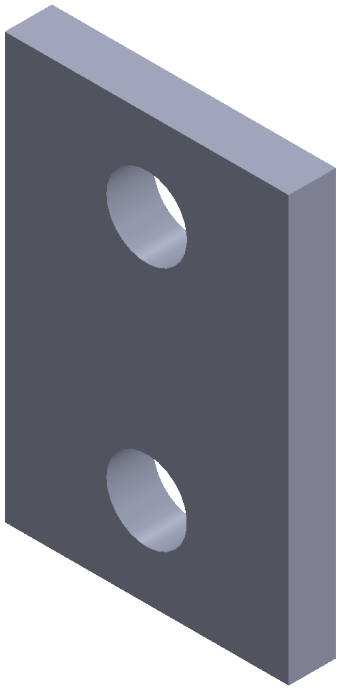
BIBEK GUPTA		
UNIVERSIDAD DE OVIEDO		
21.06.2024	Aluminium Profile Base	A4
GEWICHT:	SCALE :1:1	BLATT 1 VON 1



FRONT VIEW



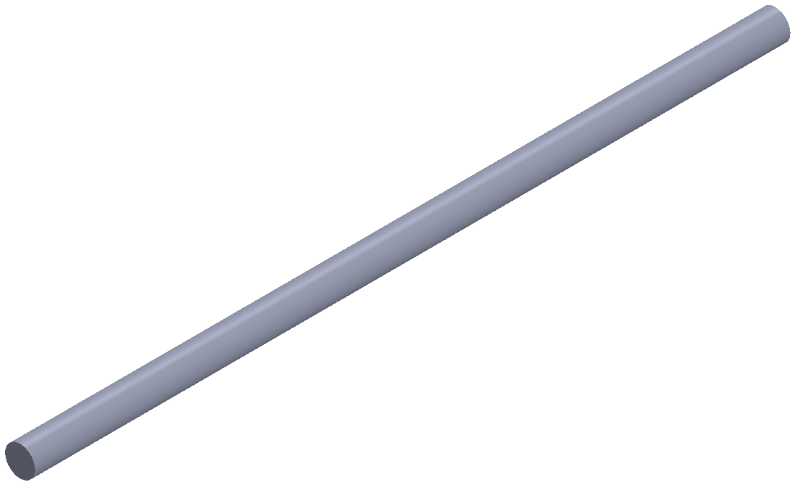
SIDE VIEW



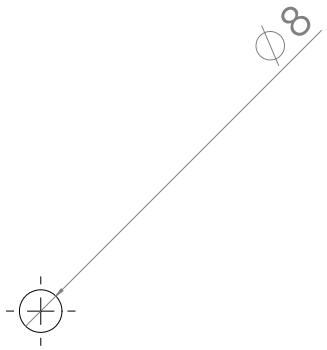
ISOMETRIC VIEW

All dimensions are in mm

BIBEK GUPTA	Universidad de Oviedo	
EU4M	Motor Filler Piece	
Master in Mechatronic Engineering		APRVD BY:
SCALE: 1:1		CHKD BY:
DATE: 25/06/2024		SHEET NO: 1 OF 1



ISOMETRIC VIEW



FRONT VIEW

SIDE VIEW

All dimensions are in mm

BIBEK GUPTA	Universidad de Oviedo	
EU4M	Conveyor Shaft	
Master in Mechatronic Engineering		APRVD BY:
SCALE: 1:1		CHKD BY:
DATE: 25/06/2024		SHEET NO: 1 OF 1

7.2. Gantt Chart

Master Thesis on Vision guided 6-axis Robotic Arm for Material Inspection on a Conveyor Line

Months/Work Details	March	April				May				June				July			
	Week 4	Week 1	Week 2	Week 3	Week 4	Week 1	Week 2	Week 3	Week 4	Week 1	Week 2	Week 3	Week 4	Week 1	Week 2	Week 3	Week 4
1. Objective Scope and Definition 2. Research and Literature Review																	
3. Hardware and Software Selection and Installation																	
4. Integration of UR3E Robot and 3D Vision Camera & Gripper																	
5. Conceptual and Detailed design of a mini conveyor line.																	
6. Material Selection and Assemblyof the conveyor.																	
7. Integration of Conveyor Line with the Robot Test bench																	
8. Algorithm Development and Simulation																	
9. Testing and Validation																	
10. Documentation and Iterative Improvement																	
11. Presentation and Defense																	

Start Date: 20th March, 2024		Work Completed	
End Date: 31st July, 2024		Work Remaining	
Student: Bibek Gupta (EU4M)			
Supervisors: 1. Prof. Juan Carlos ÁLVAREZ ÁLVAREZ 2. Prof. Miguel Ángel JOSÉ PRIETO			

7.3. Program Codes:

7.3.1. Camera_Calib_Acqisition

```

import pyrealsense2 as rs
import numpy as np
import cv2
import os

 # Set up Image directory to save the images
directory = r'C:\Users\bevec\Desktop\Images'
os.chdir(directory)

#Initial setup for Intel Reaslsense SDK
pipe = rs.pipeline()
cfg  = rs.config()

cfg.enable_stream(rs.stream.color, 640,480, rs.format.bgr8, 30)
cfg.enable_stream(rs.stream.depth, 640,480, rs.format.z16, 30)

profile = pipe.start(cfg)

try:

    while True:

        # Start camera stream
        frame = pipe.wait_for_frames()
        depth_frame = frame.get_depth_frame()
        color_frame = frame.get_color_frame()

        #Convert frames to numpy array
        depth_image = np.asanyarray(depth_frame.get_data())
        color_image = np.asanyarray(color_frame.get_data())

        #Depth Color map and Color conversion
        depth_cm = cv2.applyColorMap(cv2.convertScaleAbs(depth_image, alpha = 2), cv2.COLORMAP_JET)
        gray_image = cv2.cvtColor(color_image, cv2.COLOR_BGR2GRAY)

        width = depth_frame.get_width()
        height = depth_frame.get_height()

        cv2.imshow('rgb', color_image)
        cv2.imshow('depth', depth_cm)

        key = cv2.waitKey(1) # Assigned s key to save the image
        if key == ord('s'):
            filename = "saveimg.png"
            cv2.imwrite(filename, color_image)
        if key == ord('q'): # Assigned q key to end camera stream
            break
finally:
    pipe.stop()
    cv2.destroyAllWindows()
```

7.3.2. Camera_Calib_Calculation

```

import cv2 as cv
import numpy as np
import glob
import os
import yaml
import json
```

```
8
9
10  # Chessboard Dimensions
11  SQUARES_X = 10  # number of squares along the X axis
12  SQUARES_Y = 7  # number of squares along the Y axis
13  PATTERN = (SQUARES_X - 1, SQUARES_Y - 1)
14  SQUARE_LENGTH = 24.0  # mm, length of one square
15
16  # Get the images
17  image_dir= r'C:\Users\bevec\Desktop\Images\camcalib_new'
18
19  image_pattern = "*.png"
20  image_files = glob.glob(os.path.join(image_dir, image_pattern))
21  images = image_files
22
23  frame_size = None
24  obj_points = []  # 3D points in real world space
25  img_points = []  # 2D points in image plane.
26  objp = np.zeros((PATTERN[0] * PATTERN[1], 3), np.float32)
27  objp[:, :2] = np.mgrid[0:PATTERN[0], 0:PATTERN[1]].T.reshape(-1, 2) * SQUARE_LENGTH
28
29  for image in images:
30      # Read the image as grayscale
31      img = cv.imread(image)
32      gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
33      frame_size = gray.shape[::-1]
34
35      # Find the chessboard corners
36      ret, corners = cv.findChessboardCorners(gray, PATTERN, None)
37
38      # If found, add object points, image points (after refining them)
39      if ret == True:
40          rcorners = cv.cornerSubPix(gray, corners, (11, 11), (-1, -1), (cv.TERM_CRITERIA_EPS +
            ↪  cv.TERM_CRITERIA_MAX_ITER, 30, 0.001))
41          img_points.append(rcorners)
42          obj_points.append(objp)
43
44          # Draw and display the corners
45          cv.drawChessboardCorners(img, PATTERN, rcorners, ret)
46          cv.imshow('Original image', img)
47          directory = r'C:\Users\bevec\Desktop\Images'
48          os.chdir(directory)
49          filename = "printed.png"
50          cv.imwrite(filename, img)
51          cv.waitKey(250)
52
53  cv.destroyAllWindows()
54
55  # Get the calibrated camera parameters
56  rms_err, calib_mtx, dist_coeffs, rvecs, tvecs = cv.calibrateCamera(obj_points, img_points, frame_size,
    ↪  None, None)
57  print('Overall RMS re-projection error: %0.3f' % rms_err)
58
59  calib_mtx_list = calib_mtx.tolist()
60  dist_coeffs_list = dist_coeffs.tolist()
61  cam_data = {}
62  cam_data['Camera_Matrix'] = calib_mtx_list
63  cam_data['Dist. Coeff.'] = dist_coeffs_list
64
65  print("Camera matrix",calib_mtx_list)
66
67  json_file = r"C:\Users\bevec\Desktop\Images\cameraintrisic.json"
68  with open(json_file, 'w') as file:
69      json.dump(cam_data, file)
70
71  # Accounting Distortion effect
72  dist= [0.04, 0, 0, 0, 0.24]
73  coeff = np.array (dist)
74  h = 480; w = 640
75  newcameramtx, roi = cv.getOptimalNewCameraMatrix(calib_mtx, coeff, (w,h), 1, (w,h))
76  print(newcameramtx)
```

7.3.3. Hand_Eye_Acqisition

```
1
2
3  from robodk import robolink  # RoboDK API
4  from robodk import robomath  # Robot toolbox
5
6  import cv2 as cv
7  import numpy as np
8  from pathlib import Path
9  import json
10 from enum import Enum
11 import time
12 import os
13
14 class CameraTypes(Enum):
15     ROBODK_SIMULATED = 0
16     OPENCV_USB = 1
17
18
19 RECORD_ROBOT = True  # Record the robot pose on disk
20 RECORD_CAMERA = True  # Record the camera image on disk
21 RECORD_FOLDER = 'Hand-Eye-Data'  # Default folder to save recordings, relative to the station folder
22 ROBOT_NAME = 'UR3e'
23 #--------------------------------------
24 def get_robot(RDK: robolink.Robolink, robot_name: str):
25
26     # Retrieve the robot
27     robot_item = None
28     if robot_name:
29         robot_item = RDK.Item(robot_name, robolink.ITEM_TYPE_ROBOT)
30         if not robot_item.Valid():
31             robot_name = ''
32
33     if not robot_name:
34         robot_item = RDK.ItemUserPick("Select the robot for hand-eye", robolink.ITEM_TYPE_ROBOT)
35         if not robot_item.Valid():
36             raise
37
38     return robot_item
39
40 def record_robot(robot_item: robolink.Item, filename: str):
41
42     # Retrieve the required information for hand-eye
43     robot_data = {}
44     robot_data['joints'] = robot_item.Joints().tolist()
45     robot_data['pose_flange'] = robomath.Pose_2_TxyzRxyz(robot_item.SolveFK(robot_item.Joints()))
46
47     # Save it on disk as a JSON
48     with open(filename, 'w') as f:
49         json.dump(robot_data, f, indent=2)
50
51 def runmain():
52
53     RDK = robolink.Robolink()
54
55     robot_item = get_robot(RDK, ROBOT_NAME)
56
57     # Retrieve the folder to save the data
58     record_folder = Path(RDK.getParam(robolink.PATH_OPENSTATION)) / RECORD_FOLDER
59     record_folder.mkdir(exist_ok=True, parents=True)
60
61     # If the folder is not empty, retrieve the next ID
62     id = 0
63     ids = sorted([int(x.stem) for x in record_folder.glob('*.json') if x.stem.isdigit()])
64     if ids:
65         id = ids[-1] + 1
66
67     Image_path = Path.home() / 'Desktop' / RECORD_FOLDER
68
```

```
    while True:

        if RECORD_CAMERA:
            image_filename = os.path.join(Image_path, f'{id}.png')

            cap = cv.VideoCapture(2)
            cap.set(cv.CAP_PROP_FRAME_WIDTH, 640)
            cap.set(cv.CAP_PROP_FRAME_HEIGHT, 480)
            while True:
                ret, frame = cap.read()
                if not ret:
                    print("Error: Failed to capture image")
                    break

                cv.imshow('Camera Stream', frame)
                if cv.waitKey(1) & 0xFF == ord('s'): # Record Images of Chessboard
                    cv.imwrite(image_filename, cv.cvtColor(frame, cv.COLOR_RGB2GRAY))
                    print("ChessBoard Image saved as ",id)
                    img_Saved = True
                if cv.waitKey(1) & 0xFF == ord('q'):
                    cap.release()
                    cv.destroyAllWindows()
                    break

            if RECORD_ROBOT and img_Saved:
              robot_filename = Path(f'{record_folder.as_posix()}/{id}.json') # Records the Pose of the
              ↪  robot for the image captured
              record_robot(robot_item, robot_filename.as_posix())
              print("Robot pose no. recorded and saved as ", id)
        print ("Move to Next Position")
        break

if __name__ == '__main__':
    runmain()
```

7.3.4. Hand_Eye_Calculation

```

from robodk import robolink  # RoboDK API
from robodk import robomath  # Robot toolbox
from robodk import robodialogs  # Dialogs

import cv2 as cv
import numpy as np
import json
from pathlib import Path
from enum import Enum
import yaml

class MarkerTypes(Enum):
    CHESSBOARD = 1

HANDEYE_BOARD_TYPE = MarkerTypes.CHESSBOARD
HANDEYE_CHESS_SIZE = (10, 7)  # X/Y
HANDEYE_SQUARE_SIZE = 24  # mm
HANDEYE_MARKER_SIZE = 24  # mm

HANDEYE_FOLDER = 'Hand-Eye-Data'  # Folder to load robot poses and images for the hand-eye calibration

def pose_2_Rt(pose: robomath.Mat):
    """RoboDK pose to OpenCV pose"""
    pose_inv = pose.inv()
    R = np.array(pose_inv.Rot33())
    t = np.array(pose.Pos())
    return R, t

def Rt_2_pose(R, t):
    """OpenCV pose to RoboDK pose"""
```

```
    vx, vy, vz = R.tolist()

    cam_pose = robomath.eye(4)
    cam_pose.setPos([0, 0, 0])
    cam_pose.setVX(vx)
    cam_pose.setVY(vy)
    cam_pose.setVZ(vz)

    pose = cam_pose.inv()
    pose.setPos(t.tolist())

    return pose

def find_chessboard(img, mtx, dist, chess_size, squares_edge, refine=True, draw_img=None):

    pattern = np.subtract(chess_size, (1, 1))  # number of corners

    _img = img
    if len(img.shape) > 2:
        _img = cv.cvtColor(img, cv.COLOR_RGB2GRAY)

    # Find the chessboard's corners
    success, corners = cv.findChessboardCorners(_img, pattern)
    if not success:
        raise Exception("No chessboard found")

    if refine:
        criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)
        search_size = (11, 11)
        zero_size = (-1, -1)
        corners = cv.cornerSubPix(_img, corners, search_size, zero_size, criteria)

    if draw_img is not None:
        cv.drawChessboardCorners(draw_img, pattern, corners, success)

    if mtx is None or dist is None:
        return corners, None, None

    cb_corners = np.zeros((pattern[0] * pattern[1], 3), np.float32)
    cb_corners[:, :2] = np.mgrid[0:pattern[0], 0:pattern[1]].T.reshape(-1, 2) * squares_edge
    success, rvec, tvec = cv.solvePnP(cb_corners, corners, mtx, dist)
    if not success:
        raise Exception("No chessboard found")

    R_target2cam = cv.Rodrigues(rvec)[0]
    t_target2cam = tvec

    return corners, R_target2cam, t_target2cam

def calibrate_handeye(robot_poses, chessboard_images, camera_matrix, camera_distortion, board_type:
↪   MarkerTypes, chess_size, squares_edge: float, markers_edge: float, show_images=False):

    # Rotation part of homogeneous matrix (Tool Flange Eye to Base )
    R_gripper2base_list = []

    # Translation part of homogeneous matrix (Tool Flange Eye to Base )
    t_gripper2base_list = []

    # Rotation part of homogeneous matrix (a point in the Chessboard to the camera frame)
    R_target2cam_list = []

    # Translation part of homogeneous matrix (a point in the Chessboard to the camera frame)
    t_target2cam_list = []

    if show_images:
        WDW_NAME = 'Chessboard'
        MAX_W, MAX_H = 640, 480
        cv.namedWindow(WDW_NAME, cv.WINDOW_NORMAL)

    for i in chessboard_images.keys():
        robot_pose = robot_poses[i]
        image = chessboard_images[i]
```

```
        draw_img = None
        if show_images:
            draw_img = cv.cvtColor(image, cv.COLOR_GRAY2RGB)
        try:
            if board_type == MarkerTypes.CHESSBOARD:
                _, R_target2cam, t_target2cam = find_chessboard(image, camera_matrix,
                ↪  camera_distortion, chess_size, squares_edge, draw_img=draw_img)

        except:
            print(f'Unable to find chessboard in {i}!')
            continue

        if show_images:
            cv.imshow(WDW_NAME, draw_img)
            cv.resizeWindow(WDW_NAME, MAX_W, MAX_H)
            cv.waitKey(500)

        # Append the matrix data from all captured Images
        R_target2cam_list.append(R_target2cam)
        t_target2cam_list.append(t_target2cam)

        R_gripper2base, t_gripper2base = pose_2_Rt(robot_pose)

        R_gripper2base_list.append(R_gripper2base)
        t_gripper2base_list.append(t_gripper2base)

    if show_images:
        cv.destroyAllWindows()

    R_cam2gripper, t_cam2gripper = cv.calibrateHandEye(R_gripper2base_list, t_gripper2base_list,
    ↪  R_target2cam_list, t_target2cam_list)

    return Rt_2_pose(R_cam2gripper, t_cam2gripper)


def runmain():

    handeye_folder = Path.home() / 'Desktop' / 'Images' / 'Hand_Eye_Data'

    Cam_mtx_path = r'E:/Codes/Cam_Matrix_new.npy'
    mtx = np.load(Cam_mtx_path)

    Dist_coeff_path = r'E:/Codes/Dist_Coeff_new.npy'
    dist = np.load(Dist_coeff_path)

    # Retrieve the images and robot poses
    image_files = sorted(handeye_folder.glob('*.png'))
    robot_files = sorted(handeye_folder.glob('*.json'))

    images, poses, joints = {}, {}, {}
    for image_file, robot_file in zip(image_files, robot_files):
        if int(image_file.stem) != int(robot_file.stem):
            raise

        id = int(image_file.stem)

        image = cv.imread(image_file.as_posix(), cv.IMREAD_GRAYSCALE)
        images[id] = image

        with open(robot_file.as_posix(), 'r') as f:
            robot_data = json.load(f)
            joints[id] = robot_data['joints']
            poses[id] = robomath.TxyzRxyz_2_Pose(robot_data['pose_flange'])

    # Perform hand-eye calibration
    camera_pose = calibrate_handeye(poses, images, mtx, dist, HANDEYE_BOARD_TYPE, HANDEYE_CHESS_SIZE,
    ↪  HANDEYE_SQUARE_SIZE, HANDEYE_MARKER_SIZE)

    print(camera_pose[0]); # print each row of the hand-eye matrix
    print(camera_pose[1]);
    print(camera_pose[2]);
    print(camera_pose[3]);
```

```
172
173 if __name__ == '__main__':
174     runmain()
175
```

7.3.5. Main_Program_Conveyor

```
1  import numpy as np
2  import cv2 as cv
3  from pathlib import Path
4  import os
5  import matplotlib.pyplot as plt
6  import matplotlib.cm as cm
7
8  #importing libraries for RoboDk API
9  from robodk.robolink import *
10 from robodk.robomath import *  # Robot toolbox
11 from robodk import robodialogs  # Dialogs
12
13 # Using Robolink to access the Station Tree "targets" and "programs"
14
15 RDK = Robolink()
16 robot_item = RDK.Item('UR3e')
17 Video = RDK.Item('ImageStream')
18 ReleaseRed = RDK.Item('ReleaseRed')
19 ReleaseYellow = RDK.Item('ReleaseYellow')
20 ReleaseSize = RDK.Item('ReleaseSize')
21 ReleaseSize2 = RDK.Item('ReleaseSize2')
22 ReleaseBolt2 = RDK.Item('ReleaseBolt2')
23
24 turnH_prog = RDK.Item('Gripper_TurnH', ITEM_TYPE_PROGRAM)
25 turnV_prog = RDK.Item('Gripper_TurnV', ITEM_TYPE_PROGRAM)
26
27 Conv_ON = RDK.Item('Convey_ON', ITEM_TYPE_PROGRAM)
28 Conv_OFF = RDK.Item('Convey_OFF', ITEM_TYPE_PROGRAM)
29
30 # Set initial robot joint speed and accln
31 robot_item.setSpeedJoints(200)
32 robot_item.setAccelerationJoints(350)
33
34 # Declaration of Global Variables
35
36 Cam_mtx_path = r'E:/Codes/Cam_Matrix_new.npy'
37
38 K = np.load(Cam_mtx_path) # input camera intrinsic matrix
39
40 # Input hand-eye matrix
41
42 cam_eye_pose = [[1.000, -0.005, -0.019, -1.7660802431346987],
43                 [0.019, 0.016, 1.000, 103.34107677797118],
44                 [-0.005, -1.000, 0.017, 124.89860527439913],
45                 [0, 0, 0, 1]]
46
47 H_flange_to_eye = np.array(cam_eye_pose)
48
49 K_inv = np.linalg.inv(K)
50
51 depth = 190 # Predefined depth value
52
53 # Pose of robot during image capture/Video Capture
54
55 H_photo_pos = [ [0.000001,     0.000001,     1.000000,   365.299901],
56                 [-1.000000,     0.000001,     0.000001,  -131.050118],
57                [-0.000001,    -1.000000,     0.000001,   395],
58                 [0.000000,     0.000000,     0.000000,     1.000000 ]]
59
60 H_base_flange = np.array(H_photo_pos)
61
```

```
state = robot_item.Connect() # Verify Robot Connection State

print("Robot Connected State:", state)

robot_item.MoveJ(Video)      # Go To Video Stream target

pose = robot_item.Pose()

# Function responsible for Conveyor Start/Stop
def conveyor_cont(action):
    if action == "Start":
        Conv_ON.RunProgram()
        print("Conveyor Running")
    if action == "Stop":
        Conv_OFF.RunProgram()
        print("Conveyor Stopped")
    return None

# Function responsible for transforming 2D cordinates to 3D and robot motion
def target_set(u,v,col,orient):

    normalized_coords = np.linalg.inv(K).dot(np.array([u, v, 1]))

    x_c, y_c, z_c = normalized_coords * depth

    cam_coords = np.array([x_c, y_c, z_c, 1])

    world_coords = H_flange_to_eye.dot(cam_coords)

    target_cord = H_base_flange.dot(world_coords)

    print("Target cordinates wrt to UR3E_Base: ", target_cord[:3])

    X = target_cord[0]
    Y = target_cord[1]
    Z = target_cord[2]

    Targets_rel = [X,Y,Z]

    target = RDK.Item('PickPoint')

    pose_ref = pose

    pose_ref.setPos(Targets_rel)

    target.setPose(pose_ref)

    target.setAsCartesianTarget()

    print("Target set for centroid:",(u,v))

    robot_item.MoveJ(Video)      #Go To Video Stream target
    pause(0.25)

    if orient == 'H':
        turnV_prog.RunProgram()     #Turn Gripper Vertical
        pause(0.25)

    if orient == 'V':
        turnH_prog.RunProgram()     #Turn Gripper Horizontal
        pause(0.25)

    robot_item.setDO(10, 1)     #Gripper Open
    pause(0.25)

    robot_item.setSpeedJoints(100)  # Reduce Robot speed while moving to object
    robot_item.setAccelerationJoints(200)

    robot_item.MoveJ(target)    #MOVE TO TARGET
    pause(0.8)

    robot_item.setDO(10, 0)     #CLOSE GRIPPER
```

```
    pause(0.8)

    robot_item.setSpeedJoints(150) # Speed set back to normal
    robot_item.setAccelerationJoints(300)

    turnH_prog.RunProgram()     #Turn Gripper Horizontal back to place
    pause(0.25)

    if color == None: # Arguement passed for incorrect object size

        robot_item.MoveJ(ReleaseSize)   #Place the Object
        pause(0.25)

        robot_item.MoveJ(ReleaseSize2)   #Place the Object
        pause(0.25)

    if color == "Bolt":  #Arguement passed for deficient bolt object

        robot_item.MoveJ(Video)      #
        pause(0.25)

        robot_item.MoveJ(ReleaseBolt2)   #Place the Object
        pause(0.5)


    if color == "Red":

        robot_item.MoveJ(Video)      #Go To Video Stream target
        pause(0.25)

        robot_item.MoveJ(ReleaseRed)   #Place the Object
        pause(0.5)

    if color == "Yellow":

        robot_item.MoveJ(Video)      #MOVE TO HOME AGAIN
        pause(0.25)

        robot_item.MoveJ(ReleaseYellow)   #Place the Object
        pause(0.5)

    turnH_prog.RunProgram()     #Turn Gripper Horizontal back to place
    pause(0.25)


    robot_item.setDO(10, 1)     #Gripper Open
    pause(0.5)

    turnH_prog.RunProgram()     #Turn Gripper Horizontal
    pause(0.25)

    robot_item.MoveJ(Video)      #Go back Video Stream target
    pause(0.25)

    print("Defect Object Removed")

    return True

    # Function responsible to detect deficient bolt from ROI
def search_bolt(obj):

    hsv = cv.cvtColor(obj, cv.COLOR_RGB2HSV) #converted to HSV

    lower_silver = np.array([0,0,60]) # color mask parameters
    upper_silver = np.array([50,255,255])

    mask = cv.inRange(hsv, lower_silver, upper_silver)

    result = cv.bitwise_and(obj,obj, mask= mask)

    gray = cv.cvtColor(result, cv.COLOR_BGR2GRAY)# convert to grayscale

```

```
    equalized = cv.equalizeHist(gray) #Histogram Equalization applied

    median_blurred = cv.medianBlur(equalized, 5) # Median BLur Filter applied

    _,thresh = cv.threshold(median_blurred, 0, 100, cv.THRESH_BINARY_INV) # Thresholding

    blurr_image = cv.GaussianBlur(thresh, (9, 9), 0) # Gaussian Blur

    edges = cv.Canny(blurr_image, 10,20) # Canny Edge Detection

    cv.imshow('roi_canny', edges)

    contours, _ = cv.findContours(edges, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)

    print("Number of contours", len(contours))
    circles = []

    #This loop iterates over all the contours extracted from the image
    for contour in contours:
        if cv.contourArea(contour)> 500 and cv.contourArea(contour)< 1000:

            (x, y), radius = cv.minEnclosingCircle(contour) # Circle Approximation
            if radius > 12 and radius < 20:  # Adjust the threshold range as per radius of bolts
            # Draw the circle
                center = (int(x), int(y))
                circles.append((int(x), int(y)))

                radius = int(radius)
                print("Radius",radius)
                cv.circle(result, center, radius, (0, 255, 0), 2)

            cv.imshow('roi_approx_circles', result)

    bolts_found = len(circles)
    print("No of Bolts:", bolts_found)

    return bolts_found

# Program Configuration setup
Video_ACK_PRCS = True
Conv_Move = True
Robot_Move = True
Program_RedYellow = False
Program_Object_Size = False
Program_Missing_Bolt = True

if Program_Missing_Bolt == True:
    depth = 165 #Since Height of the object is more for this program mode.
else:
    depth = 190

if Conv_Move: # Initial Start of Conveyor
    act = "Start"
    conveyor_cont(act)

if Video_ACK_PRCS: # Turn on Camera if set True

    cap = cv.VideoCapture(2)
    cap.set(cv.CAP_PROP_FRAME_WIDTH, 640)
    cap.set(cv.CAP_PROP_FRAME_HEIGHT, 480)
    key = cv.waitKey(1) & 0xFF

    while True:
        ret, frame = cap.read() # Read Image frames
        if not ret:
            print("Error: Failed to connect")
            break
        cv.imshow('Camera video stream', frame)

        if cv.waitKey(1) & 0xFF == ord('q'):
            print("Camera Stream Closed")
            act = "Stop"
```

```
278             conveyor_cont(act)
279             cap.release()
280             cv.destroyAllWindows()
281             break
282 
283         image_org = cv.cvtColor(frame, cv.COLOR_RGB2BGR)
284 
285         image_org_clone = image_org.copy()
286 
287         #Detection Window parameters defined
288         x1, y1 = 180,50  # top-left corner
289         x2, y2 = 500, 430  # bottom-right corner
290 
291         image_cropped = image_org[y1:y2, x1:x2]
292 
293         cv.imshow('Cropped', image_cropped)
294 
295         blurr_image = cv.GaussianBlur(image_cropped, (5, 5), 0)
296 
297         hsv = cv.cvtColor(image_cropped, cv.COLOR_BGR2HSV)
298 
299         #Color Mask operation to extract foreground objects
300         lower_blue = (32, 50, 80)
301         upper_blue = (140, 255, 255)
302 
303         mask_blue = cv.inRange(hsv, lower_blue, upper_blue)
304 
305         mask_foreground = cv.bitwise_not(mask_blue)
306 
307         image_rgb = cv.cvtColor(image_cropped, cv.COLOR_BGR2RGB)
308 
309         background = np.ones_like(image_rgb, np.uint8) * 255
310 
311         foreground = cv.bitwise_and(image_rgb, image_rgb, mask=mask_foreground)
312 
313         result = cv.add(foreground, cv.bitwise_and(background, background, mask=mask_blue))
314 
315         cv.imshow('Foreground Objects of Interest', result)
316 
317         gray = cv.cvtColor(result, cv.COLOR_RGB2GRAY)
318 
319         _,thresh = cv.threshold(gray, 250, 255, cv.THRESH_BINARY_INV)
320 
321         cv.imshow('Threshold',thresh)
322 
323         blurr_image = cv.GaussianBlur(thresh, (5, 5), 0)
324 
325         edges = cv.Canny(blurr_image, 10, 20)
326 
327         cv.imshow('Canny', edges)
328 
329         contours, _ = cv.findContours(edges, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)
330 
331         # print("Number of contours", len(contours))
332 
333         centroids = []
334         centroidsred = []
335         centroidsyellow = []
336         centroidssmall =[]
337         centroidbolt = []
338         orientation = ["H", "V"]
339 
340         i = 0
341 
342         for contour in contours:
343             epsilon = 0.08 * cv.arcLength(contour, True)
344             approx = cv.approxPolyDP(contour, epsilon, True)
345 
346             #Extract countours with four edges and filter out minor contour areas
347             if len(approx) == 4 and cv.contourArea(contour) > 4000:
348 
349                 x, y, w, h = cv.boundingRect(contour) #Object Edge Cordinate in the image
```

```
350
351             aspect_ratio = w / float(h) # Aspect ratio calulation
352
353             if aspect_ratio > 1:  # Determine Orientation
354                 ort = orientation[0]
355             else:
356                 ort = orientation[1]
357
358             print(cv.contourArea(contour))
359
360             roi = image_cropped[y:y+h, x:x+w] # Extract the region of interest
361
362             cv.imshow('roi_color', roi)
363
364             average_color_rgb = cv.mean(roi) #RGB value calculation
365
366             print("Average color (RGB):", average_color_rgb)
367
368             print ("Edges:", len(approx))
369
370             color_label = ["Yellow", "Red", "Noi"]
371
372             temp_color = 0 # Color Defining Criteria
373
374             if (average_color_rgb[0] >=100) and (average_color_rgb[1] >= 100) and
                ↪  (average_color_rgb[2] <= 75) :
375                 cv.putText(image_cropped, color_label[0], (x, y - 10), cv.FONT_HERSHEY_SIMPLEX,
                    ↪  0.9, (255, 255, 255), 2)
376                 temp_color = "Yellow"
377             if (average_color_rgb[0] >=100) and (average_color_rgb[1] <= 75) and
                ↪  (average_color_rgb[2] <= 75) :
378                 cv.putText(image_cropped, color_label[1], (x, y - 10), cv.FONT_HERSHEY_SIMPLEX,
                    ↪  0.9, (255, 255, 255), 2)
379                 temp_color = "Red"
380
381             M = cv.moments(contour)
382
383             if M["m00"] != 0:
384                 # Calculate x, y coordinate of centroid
385                 cX = int(M["m10"] / M["m00"])
386                 cY = int(M["m01"] / M["m00"])
387                 centroids.append((cX, cY))
388                 cv.drawContours(image_cropped, [contour], -1, (0, 255, 0), 2)
389                 cv.circle(image_cropped, (cX, cY), 7, (0, 0, 255), -1)
390                 cv.putText(image_cropped, f"({cX}, {cY})", (cX - 20, cY - 20),
                    ↪  cv.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 2)
391
392                 cv.imshow("Centroids of Blocks", image_cropped)
393
394                 if Robot_Move:
395
396                     if Program_RedYellow: #if set True
397
398                         if temp_color == "Yellow":
399                             # Stop Conveyor
400                             act = "Stop"
401                             conveyor_cont(act)
402
403                             print("Defect Detected: Yellow Object")
404
405                             centroidsyellow.append((cX + 180 , cY + 50, ort))
406
407                             if len(centroidsyellow)>0:
408                                 for i in range(len(centroidsyellow)):
409                                     px,py,o = centroidsyellow[i]
410                                     color = "Yellow"
411                                     status = target_set(px, py, color, o)
412
413                             if status:
414                                 centroidsyellow = []
415                                 act = "Start"
416                                 conveyor_cont(act)
```

```
                    if temp_color == "Red":
                        # Stop Conveyor
                        act = "Stop"
                        conveyor_cont(act)

                        print("Defect Detected: Red Object")

                        centroidsred.append((cX + 180 , cY + 50, ort))

                        if len(centroidsred)>0:
                            for i in range(len(centroidsred)):
                                px,py,o = centroidsred[i]
                                color = "Red"
                                status = target_set(px, py, color, o)

                        if status:
                            act = "Start"
                            conveyor_cont(act)

                if Program_Object_Size: #if set True

                        area = cv.contourArea(contour)
                        print("Countour Area:", area)

                        if area>=8000 and area<=16000: # Object Size criteria definition
                                # Stop Conveyor
                                act = "Stop"
                                conveyor_cont(act)

                                print("Defect Detected: Small Object")

                                centroidssmall.append((cX + 180 , cY + 50, ort))

                                if len(centroidssmall)>0:
                                    for i in range(len(centroidssmall)):
                                        px,py,o = centroidssmall[i]
                                        color = None
                                        status = target_set(px, py, color, o)

                                if status:
                                    act = "Start"
                                    conveyor_cont(act)

                if Program_Missing_Bolt: #if set True

                    if len(centroids)>0:
                        obj_det = True

                    #Extension of ROI for accurate bolt detection

                    a1, b1 = 180 + x - 20 ,50 + y - 20  # top-left corner
                    a2, b2 = 180 + x - 20 + w + 60 ,50 + y - 20 + h + 60  # bottom-right corner

                    image_crop_bolt = image_org_clone[b1:b2, a1:a2]

                    cv.imshow('roi_bolt_cropped', image_crop_bolt)

                    area = cv.contourArea(contour)
                    # print("Countour Area:", area)

                    if area>=20500 and area<=22000:

                        bolt = search_bolt(image_crop_bolt)

                        if obj_det == True and bolt in range(0,4):
                            act = "Stop"
                            conveyor_cont(act)
                            print("Defect Detected: Missing Bolt/s:", 4 -bolt)
                            print("Removal in progress")
                            centroidbolt.append((cX + 180 , cY + 50, ort))
```

```
                                    if len(centroidbolt)>0:
                                        for i in range(len(centroidbolt)):
                                            px,py,o = centroidbolt[i]
                                            color = "Bolt"
                                            o= "V"
                                            status = target_set(px, py, color, o)

                                    if status:
                                        act = "Start"
                                        conveyor_cont(act)

```