# SEM4405 Robotics Project
# Wireless Control Smart Mobile Manipulator

**Systems Engineering (ElectroMechanical Systems)**
**Module: SEM4405 Robotics**
**Academic Year: AY2023/2024, Trimester 2**
**Academic Supervisor: Dr Liaw Hwee Choo**
**Group 5**


| Team Members | Student ID |
|---|---|
| Nur Fatin Farzana Binte Mohd Hazeem | 2001381 |
| Muhammad As-Siddique Bin Mohd Rashid | 2001124 |
| Muhammad Noraffiq Bin Mohd Roslee | 2001503 |
| Muhammad Hafizin Bin Mazli | 2000965 |

# Table of Content


Table of Content ---- 2
1.Introduction ---- 3
1.1 Project Objectives ---- 3
1.2 Project Capabilities ---- 3
2. Overall implementations ---- 4
2.1 Hardware ---- 4
2.2 Software ---- 4
2.2.1 DH Parameters and Robot Equations ---- 4
2.2.1.1 Link frame assignment ---- 5
2.2.1.2 Robot Constant Parameters ---- 5
2.2.1.3 Symbolic equations of the configuration coordinate ---- 5
2.2.1.4 Symbolic equations of the forward and inverse kinematics ---- 6
2.2.3 Host control GUI (C#) ---- 8
2.2.4 Implemented Features ---- 13
2.2.4.1 Object Detection ---- 13
2.2.4.2 Color Detection ---- 15
2.2.4.3 Shape Detection ---- 17
2.2.4.4 Canny Edge Detection ---- 20
2.2.4.5 Sobel Edge Detection ---- 23
2.2.4.6 Draw square box ---- 25
2.2.4.7 Pick and Place of an object ---- 27
4.Conclusion ---- 28
4.1 Summary of achievements ---- 28
4.2 Challenges Faced ---- 28
4.3 Lessons Learnt ---- 28

# 1.Introduction

This group project involves the development of a wireless-controlled intelligent mobile manipulator, serving as a practical application of the knowledge and competencies attained in this module for the implementation of a sophisticated robotic system. This system is expected to possess the capability to discern, analyze, and address challenges through algorithmic solutions in order to accomplish designated tasks efficiently.

The envisioned robot is required to exhibit the following functionalities:

- Capable of grasping and releasing objects.
- Equipped with camera-based capabilities to recognize color, shape, objects, and edges.
- Proficient in utilizing a robotic arm to execute tasks such as drawing straight lines and squares.
- Incorporation of a user-friendly graphical interface enabling remote management of the intelligent robot system via wireless connectivity to a computer.

## 1.1 Project Objectives

The objectives of this project encompass:

- Application of acquired robotics knowledge from lectures to actualize an experimental intelligent robotic system endowed with multiple capabilities.
- Proficiency in identifying, formulating, and resolving challenges pertinent to the implemented smart mobile manipulator.
- Practical demonstration of robotic functionalities.
- Analysis and interpretation of computed and measured data to actualize purposeful robotic manipulation functions and tasks.
- Utilization of acquired techniques, skills, and contemporary engineering tools to materialize the proposed robotic tasks.

## 1.2 Project Capabilities

The intended capabilities for this project include:

- Establishment of wireless communication between the control PC and the smart mobile manipulator.
- Creation of a graphical user interface (GUI) to exhibit the robot's status and information, facilitate user input, and execute commands.
- Demonstration of functionalities such as forward and inverse kinematics, as well as trajectory following.
- Utilization of the camera to capture images and apply image processing techniques such as color, object, shape, and edge detection.
- Implementation of object sensing and estimation of distance between the robot and detected objects.
- Execution of robotic manipulator tasks to successfully grasp and release objects.

# 2. Overall implementations

## 2.1 Hardware

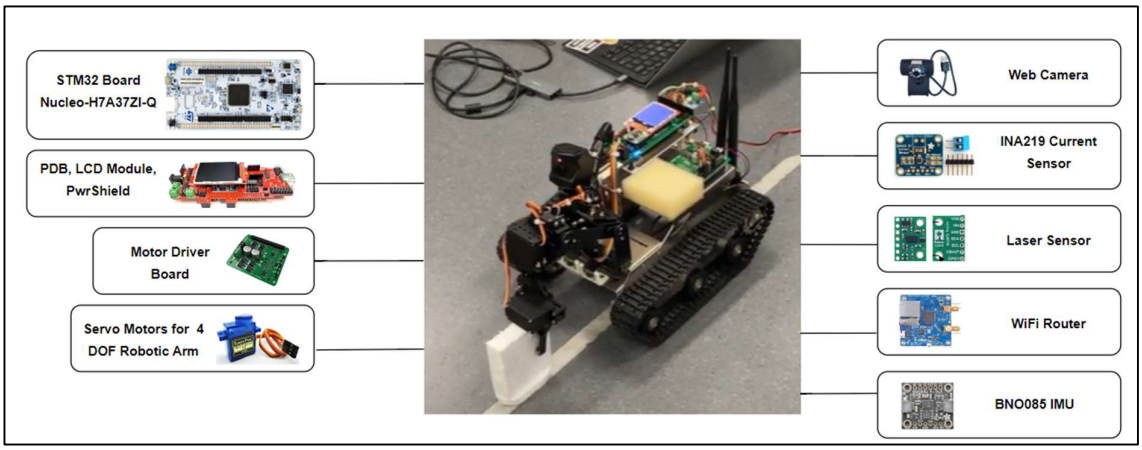



*Figure 1: Breakdown of Hardware Components*

## 2.2 Software

### 2.2.1 DH Parameters and Robot Equations

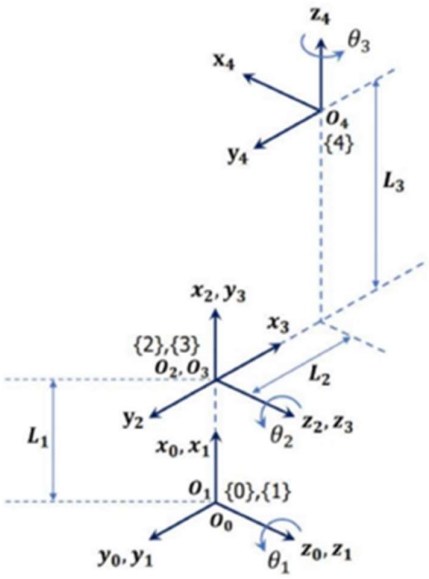



*Figure 2: Sketch of links of the Smart Manipulator Robot*

Figure 2.4 shows a sketch draw of the links of the Smart Manipulator Robot. This robot has four degrees of freedom. From the sketch we can see that, there are three links L1, L2, L3 and three joint angles $\theta 1$, $\theta 2$, $\theta 3$.

### 2.2.1.1 Link frame assignment

**Q1. The diagram of the link-frame assignment for the robot manipulator is shown in Figure 1. Tabulate a table of D-H parameters based on the link-frame assignment.**

| | Link i | | | |
|---|---|---|---|---|
| D-H Parameters | 1 | 2 | 3 | 4 |
| $\propto_i$ | 0 | 0 | 0 | -90 |
| $b_i$ | 0 | $L_1$ | 0 | $L_2$ |
| $\theta_i$ | $\theta_1$ | $\theta_1$ | -90 | $\theta_3 + 90$ |
| $d$ | 0 | 0 | 0 | $L_3$ |

*Figure 3: Modified D-H parameters of the robot.*

### 2.2.1.2 Robot Constant Parameters

**Q2. The robot constant parameters are given as follows: L1 = 95 mm L2 = 28 mm L3 = 155 mm**

### 2.2.1.3 Symbolic equations of the configuration coordinate

**Q4. Determine the symbolic configuration coordinate transformation of the robot, 0T4. For 0T4, express the elements of rotation matrix as:**

$$0_{T4} = \begin{bmatrix} -s_{12}s_3 & -s_{12}s_3 & c_{12} & L_3c_{12} + L_2c_{12} + L_1c_1 \\ c_{12}s_3 & c_{12}c_3 & s_{12} & L_3s_{12} - L_2c_{12} + L_1s_1 \\ -c_3 & s_3 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where $c_i = \cos(\theta_i), s_i = sin(\theta_i), s_{12} = sin(\theta_1 + \theta_2), c_{12} = \cos(\theta_1 + \theta_2)$

**Rotation Matrix**

$r_{11} = -sin(\theta_1 + \theta_2)\, sin\,(\theta_3)$

$r_{12} = -sin(\theta_1 + \theta_2) \cos\,(\theta_3)$

$r_{13} = \cos(\theta_1 + \theta_2)$

$r_{21} = \cos(\theta_1 + \theta_2)\ \ sin\,(\theta_3)$

$r_{22} = \cos(\theta_1 + \theta_2)\ \ \cos\,(\theta_3)$

$r_{23} = sin(\theta_1 + \theta_2)$

$r_{31} = -\cos\,(\theta_3)$

$r_{32} = sin\,(\theta_3)$

$r_{33} = 0$

**Position Vector**

$p_x = L_3 \cos(\theta_1 + \theta_2) + L_2 sin(\theta_1 + \theta_2) + L_1 \cos(\theta_1)$

$p_y = L_3 \sin(\theta_1 + \theta_2) - L_2 cos(\theta_1 + \theta_2) + L_1 \sin(\theta_1)$

$p_z = 0$

The rotation matrix elements and the position vectors which can be inferred from the coordinate transformation matrix are shown above.

### 2.2.1.4 Symbolic equations of the forward and inverse kinematics

Q5. The quaternion can be presented in the form of q = q0 + q1 i + q2 j + q3 k, where q0, q1, q2, and q3 are the real numbers; and i, j, k are the basis vectors. Obtain the symbolic expressions of q0, q1, q2, and q3 in terms of the elements of rotation matrix shown in Q4. In week 2 lecture, discussion of the information from CH Robotics, the elements of quaternion are a = q0, b = q1, c = q2, and d = q3.

$$q0 = {}^{1}/_{4}\,\pm\sqrt{(r11 + r22 + r33 + 1)^2 + (r32 - r23)^2 + (r13 - r31)^2 + (r21 - r12)^2}$$

$$q1 = {}^{1}/_{4}\,\pm\sqrt{(r32 - r23)^2 + (r11 - r22 - r33 + 1)^2 + (r21 + r12)^2 + (r31 + r13)^2}$$

$$q2 = {}^{1}/_{4}\pm\sqrt{(r13 - r31)^2 + (r2 + r12)^2 + (r22 - r11 - r33 + 1)^2 + (r32 + r23)^2}$$

$$q3 = {}^{1}/_{4}\,\pm\sqrt{(r21 - r12)^2 + (r31 + r13)^2 + (r32 + r23)^2 + (r33 - r11 - r22 + 1)^2}$$

Q6. Determine the equations of ZYX Euler angles, i.e., obtain the symbolic expressions of yaw, pitch, and roll angles.

If Abs(r31 > 1) < Epsilon
    Roll = 0
    Pitch = $-\ sin^{-1}(r31)$
    If r31>0 then
        Yaw = $atan2(\frac{-r23}{-r13})$

else
    Roll = $atan2(\frac{r32}{r33})$
    Yaw = $atan2(\frac{r21}{r11})$
    If (r11 >= r21 && r11 >= r32 && r11 >= r33)
        Pitch = $atan2(\frac{-r31 * Cos(Yaw)}{r11})$
    If (r21 >= r11 && r21 >= r32 && r21 >= r33)
        Pitch = $atan2(\frac{-r31 * Cos(Yaw)}{r21})$
    If (r32 >= r21 && r32 >= r11 && r32 >= r33)
        Pitch = $atan2(\frac{-r31 * Sin(Roll)}{r32})$
    If (r33 >= r21 && r33 >= r32 && r33 >= r11)
        Pitch = $atan2(\frac{-r31 * Cos(Roll)}{r33})$

The elements r11, r12, r13, r21, r22, r23, r31, r32, r33 in the rotation matrix represent various components. According to 'Cayley's Method', provided q0 is positive, we can ensure consistency among the other Euler parameters by assigning the signs of (r32-r23), (r12-r31), and (r21-r12) to q1, q2, and q3 respectively.

Q7. Determine the symbolic equations of inverse kinematics, i.e., obtain θ1 and θ2 in terms of px, py, and robot constants.

$$\theta 2 = atan2\left(\frac{L2}{L3}\right) + atan2\left(\frac{\pm\sqrt{L3^2 + L2^2 - \frac{Px^2 + Py^2 - L1^2 - L2^2 - L3^2}{2L1}}}{\frac{Px^2 + Py^2 - L1^2 - L2^2 - L3^2}{2L1}}\right)$$

$$\theta 1 = atan2\left(\frac{(L1 + c2L3 + s2L2)Py - (s2L3 - c2L2)Px}{(L1 + c2L3 + s2L2)Px + (s2L3 - c2l2)Py}\right)$$

where $s\theta 2 = sin\theta 2, c\theta 2 = cos\theta 2$

## 2.2.3 Host control GUI (C#)

The GUI host control, developed using Microsoft Visual Studio, provides an intuitive interface for commanding the vehicle to execute its robotic tasks. Initially, the project was supplied with a basic GUI for manual robot control, which was subsequently extensively revised. These revisions included integrating logic and calculations for robotic arm kinematics, image processing functionalities, pick-and-place operations, among others. Additionally, modifications were made to enable control over robot actuation and user input management. The host control program is also responsible for receiving sensor data and encoder readings from the robot software.

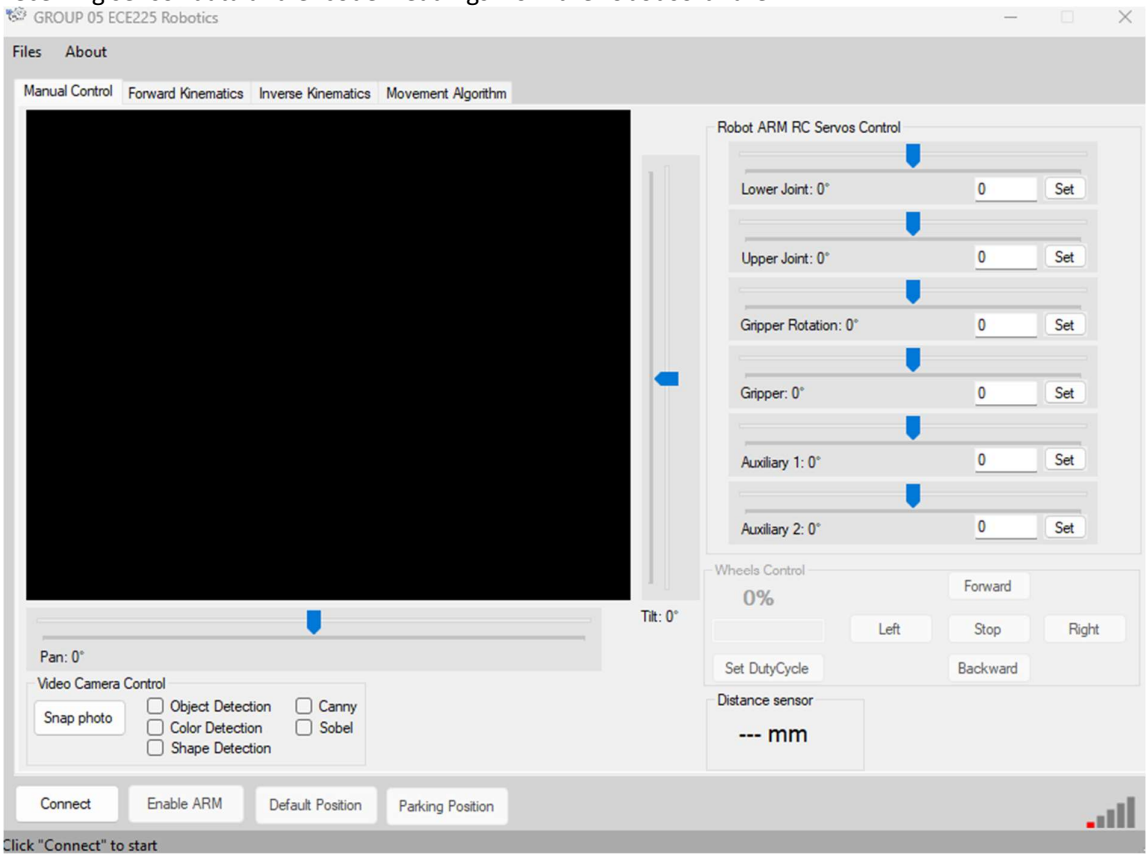



*Figure 3: GUI (First Tab)*

The primary tab of the GUI, depicted in Figure 2.15, serves as the central page housing manual controls situated on the right-hand side of the interface. These controls allow for the adjustment of angles for the 4 Degree of Freedom (DOF) manipulator and basic movement commands for the robot, such as forward and backward motions. On the left side of the window, the captured video feed from the robot's camera is showcased. Below this display, checkboxes are provided for selecting the desired image processing techniques to apply to the captured video. Additionally, sensor readings are presented in the bottom-right corner of the window.

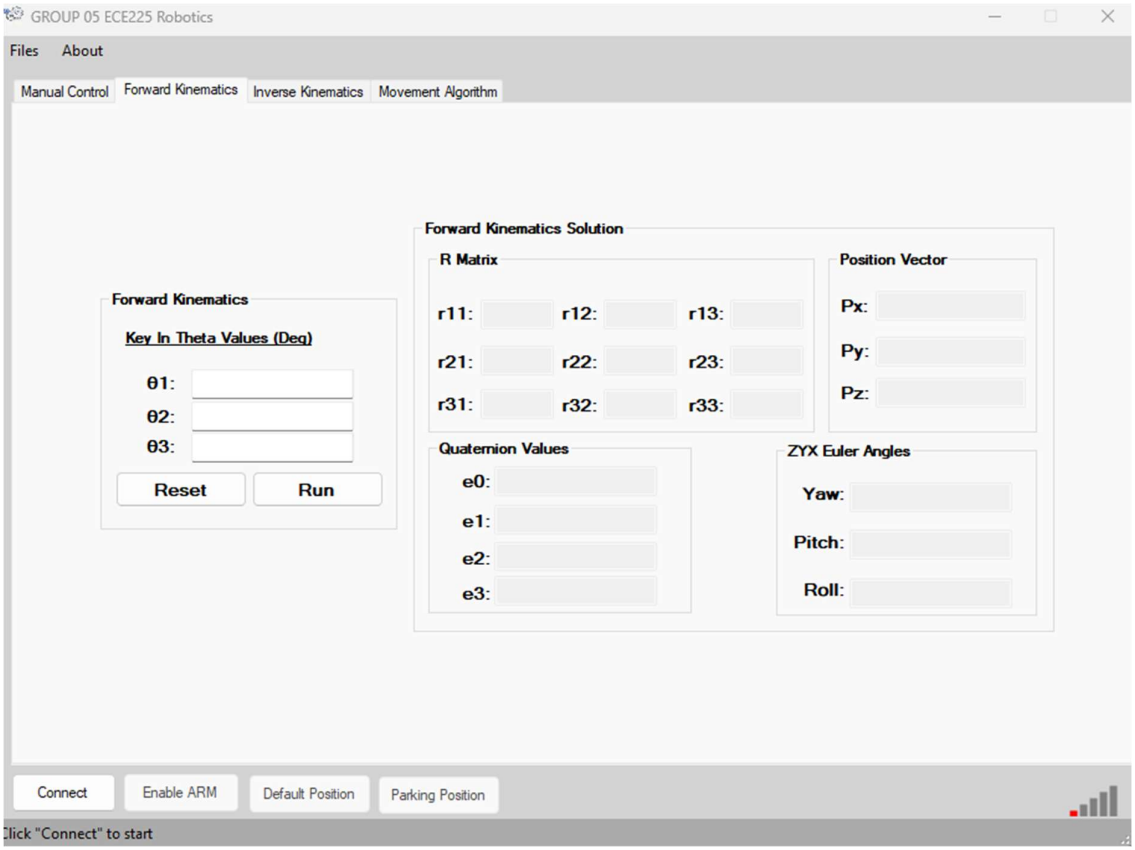



*Figure 4: GUI (Second Tab - Forward Kinematics)*

```
// Convert angle from textbox to radians' local variable
q1 = UpdateRADValue(Convert.ToDouble(tbTeta1.Text));
q2 = UpdateRADValue(Convert.ToDouble(tbTeta2.Text));
q3 = UpdateRADValue(Convert.ToDouble(tbTeta3.Text));

// Convert angle from textbox to number
double  q1_deg = Convert.ToDouble(tbTeta1.Text);
double  q2_deg = Convert.ToDouble(tbTeta2.Text);
double  q3_deg = Convert.ToDouble(tbTeta3.Text);

//Set values for R matrix textboxes
tbR11.Text = (-Math.Sin(q1 + q2) * Math.Sin(q3)).ToString("0.000");
tbR12.Text = (-Math.Sin(q1 + q2) * Math.Cos(q3)).ToString("0.000");
tbR13.Text = (Math.Cos(q1 + q2)).ToString("0.000");
tbR21.Text = (Math.Cos(q1 + q2) * Math.Sin(q3)).ToString("0.000");
tbR22.Text = (Math.Cos(q1 + q2) * Math.Cos(q3)).ToString("0.000");
tbR23.Text = (Math.Sin(q1 + q2)).ToString("0.000");
tbR31.Text = (-Math.Cos(q3)).ToString("0.000");
tbR32.Text = (Math.Sin(q3)).ToString("0.000");
tbR33.Text = 0.ToString("0.000");

//Set values for XYZ position textboxes
tbPx.Text = ((Link2 * Math.Sin(q1 + q2)) + (Link3 * Math.Cos(q1 + q2)) + (Link1 * Math.Cos(q1))).ToString("0.000");
tbPy.Text = ((Link3 * Math.Sin(q1 + q2)) - (Link2 * Math.Cos(q1 + q2)) + (Link1 * Math.Sin(q1))).ToString("0.000");
tbPz.Text = 0.ToString("0.000");

//Store R matrix values in local variables
setquatval();

//Update Quaternion values
e0 = (float)((Math.Sqrt((Math.Pow(r11 + r22 + r33 + 1, 2)) + (Math.Pow(r32 - r23, 2)) + (Math.Pow(r13 - r31, 2)) + (Math.Pow(r21 - r12, 2)))) / 4);
e1 = (float)((Math.Sqrt((Math.Pow(r32 - r23, 2)) + (Math.Pow(r11 - r22 - r33 + 1, 2)) + (Math.Pow(r21 + r12, 2)) + (Math.Pow(r31 + r13, 2)))) / 4);
e2 = (float)((Math.Sqrt((Math.Pow(r13 - r31, 2)) + (Math.Pow(r21 + r12, 2)) + (Math.Pow(r22 - r11 - r33 + 1, 2)) + (Math.Pow(r32 + r23, 2)))) / 4);
e3 = (float)((Math.Sqrt((Math.Pow(r21 - r12, 2)) + (Math.Pow(r31 + r13, 2)) + (Math.Pow(r32 + r23, 2)) + (Math.Pow(r33 - r11 - r22 + 1, 2)))) / 4);
```

*Figure 5: Forward Kinematics C# code*

```
double Yaw = 0;
double Pitch = 0;
double Roll = 0;
if (Math.Abs(Math.Abs(r31) - 1) < Double.Epsilon)
{
    Roll = 0;
    if (r31 < 0)
    {
        Yaw = Math.Atan2(r23, r13);
    }
    else
    {
        Yaw = Math.Atan2(-r23, -r13);
    }
    Pitch = -(Math.Asin(r31));
}
else
{
    Roll = Math.Atan2(r32, r33);
    Yaw = Math.Atan2(r21, r11);
    if (r11 >= r21 && r11 >= r32 && r11 >= r33)
    {
        Pitch = Math.Atan2(-r31 * Math.Cos(Yaw), r11);
    }
    if (r21 >= r11 && r21 >= r32 && r21 >= r33)
    {
        Pitch = Math.Atan2(-r31 * Math.Sin(Yaw), r21);
    }
    if (r32 >= r21 && r32 >= r11 && r32 >= r33)
    {
        Pitch = Math.Atan2(-r31 * Math.Sin(Roll), r32);
    }
    if (r33 >= r21 && r33 >= r32 && r33 >= r11)
    {
        Pitch = Math.Atan2(-r31 * Math.Cos(Roll), r33);
    }
}
```

*Figure 5: Forward Kinematics C# code*

The subsequent tab (as illustrated in Figure 4) serves as the input interface and display area for forward kinematics results. On the left-hand side of the window, users can input joint angles, and upon pressing the "run" button, computed rotation matrix elements, position vectors, quaternions, and ZYX Euler angles will be presented. The code snippet executed upon pressing the "run" button for forward kinematics is depicted in Figure 5. Clicking the "reset" button will reset all values to zero.

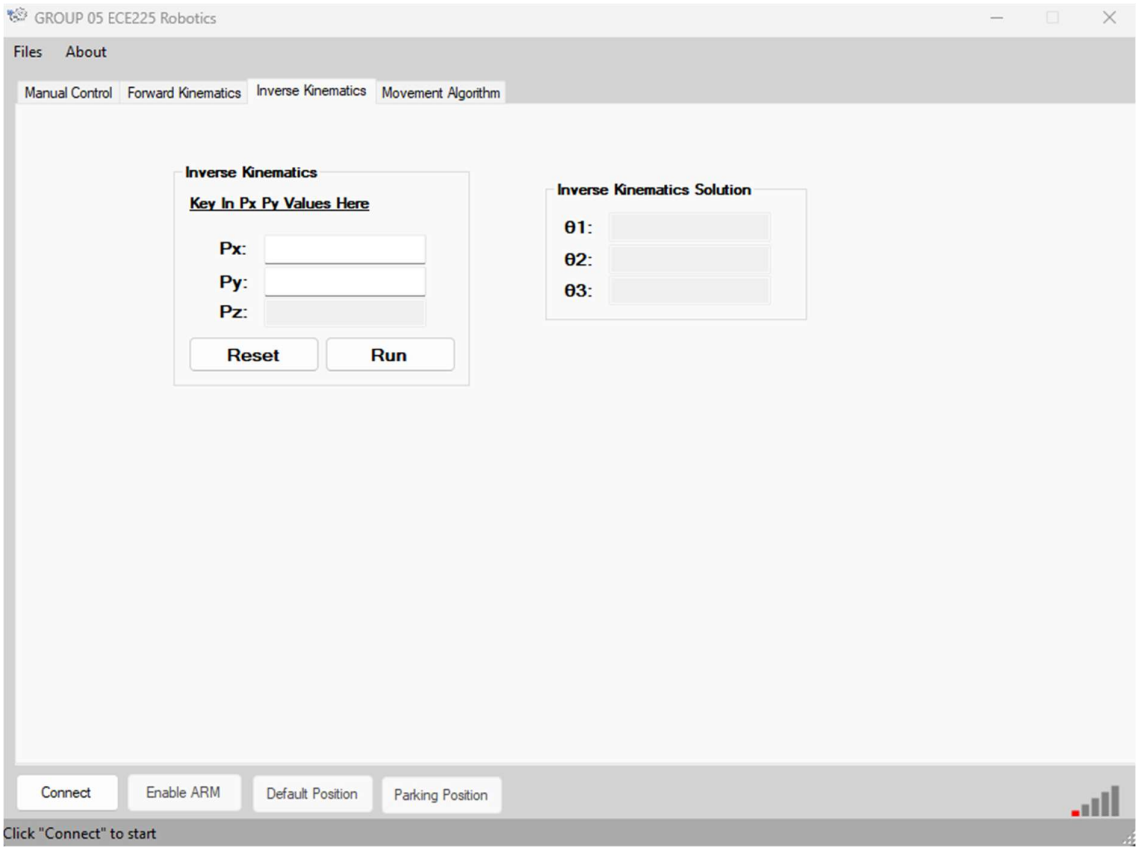



*Figure 7: GUI (Third Tab - Inverse Kinematics)*

```
double Link1 = 95;
double Link2 = 28;
double Link3 = 155;

double px1 = Convert.ToDouble(tbPxInv.Text);
double py1 = Convert.ToDouble(tbPyInv.Text);

double a = (2 * Link3 * Link1);
double b = (2 * Link2 * Link1);
double c = Math.Pow(px1, 2) + Math.Pow(py1, 2) - Math.Pow(Link3, 2) - Math.Pow(Link2, 2) - Math.Pow(Link1, 2);

double xx = Math.Atan2(b, a);
double yy = Math.Atan2(Math.Sqrt(Math.Pow(a, 2) + Math.Pow(b, 2) - Math.Pow(c, 2)), c);
double sq = Math.Sqrt(Math.Pow(a, 2) + Math.Pow(b, 2) - Math.Pow(c, 2));
double theta2_1 = Math.Atan2(b, a) + Math.Atan2(Math.Sqrt(Math.Pow(a, 2) + Math.Pow(b, 2) - Math.Pow(c, 2)), c);
double theta2_2 = Math.Atan2(b, a) + Math.Atan2(-Math.Sqrt(Math.Pow(a, 2) + Math.Pow(b, 2) - Math.Pow(c, 2)), c);

double a_1 = (Link3 * Math.Cos(theta2_1)) + (Link2 * Math.Sin(theta2_1) + Link1);
double b_1 = (Link3 * Math.Sin(theta2_1)) - (Link2 * Math.Cos(theta2_1));
double c_1 = px1;
double d_1 = py1;
double theta1_1 = Math.Atan2(((a_1 * d_1) - (b_1 * c_1)), ((a_1 * c_1) + (b_1 * d_1)));

double a_2 = (Link3 * Math.Cos(theta2_2)) + (Link2 * Math.Sin(theta2_2) + Link1);
double b_2 = (Link3 * Math.Sin(theta2_2)) - (Link2 * Math.Cos(theta2_2));
double c_2 = px1;
double d_2 = py1;
double theta1_2 = Math.Atan2(((a_2 * d_2) - (b_2 * c_2)), ((a_2 * c_2) + (b_2 * d_2)));

double t1_1 = ((theta1_1 * (180 / Math.PI)));
double t2_1 = ((theta2_1 * (180 / Math.PI)));

double t1_2 = ((theta1_2 * (180 / Math.PI)));
double t2_2 = ((theta2_2 * (180 / Math.PI)));
```

*Figure 8: Inverse Kinematics C# code*

The subsequent tab is dedicated to computing inverse kinematics, depicted in Figure 7. Upon inputting values on the left side (Px and Py values), users can click the "run" button to execute the inverse kinematics function. Pressing the "reset" button will revert all values to zero. A segment of the code pertaining to inverse kinematics is provided in Figure 8.

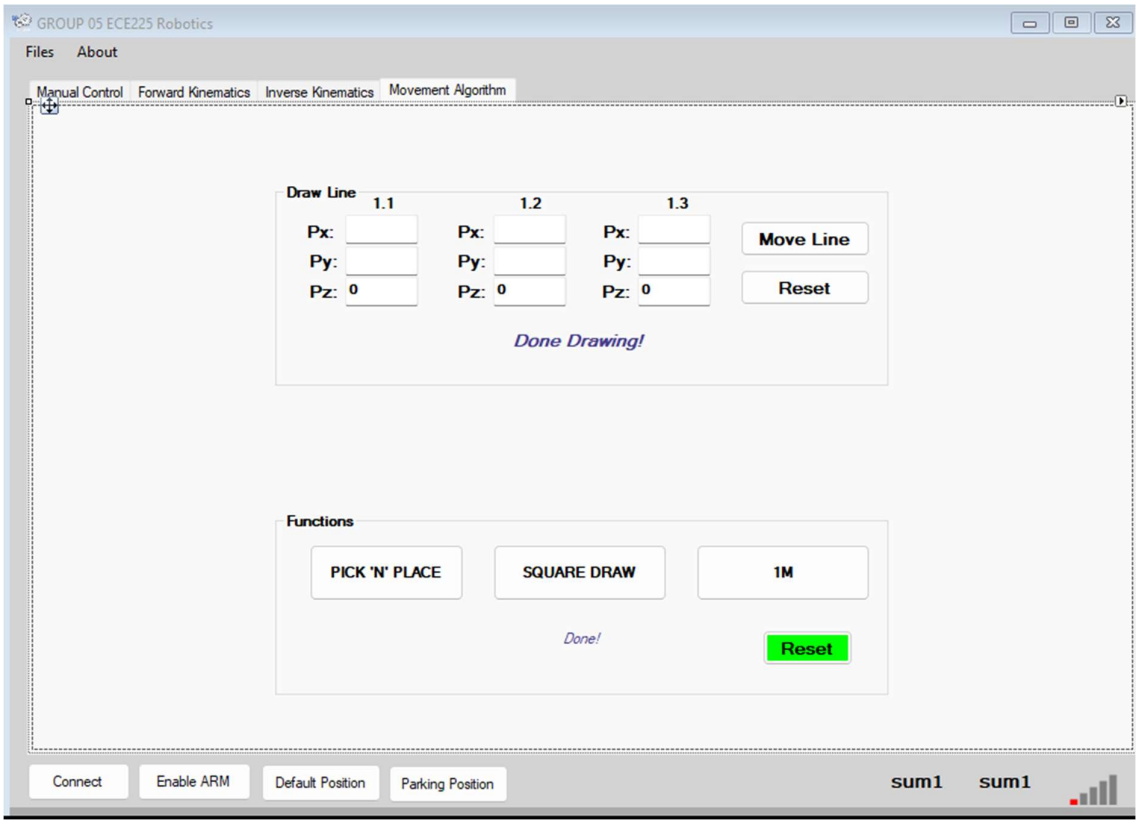



*Figure 9: GUI (Final Tab)*

The last tab, illustrated in Figure 9, houses a range of implemented functions including drawing lines, picking and placing objects, and drawing squares.

## 2.2.4 Implemented Features

### 2.2.4.1 Object Detection

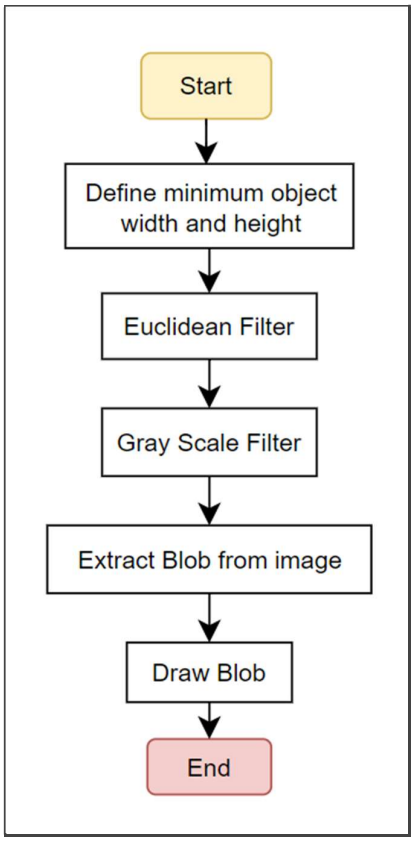



*Figure 10: Flow chart for object detection*

```
1 reference
private void FindObjects(Bitmap image, Graphics g )
{
    BlobCounter blobCounter = new BlobCounter();
    blobCounter.MinWidth = 50;
    blobCounter.MinHeight = 50;
    blobCounter.FilterBlobs = true;
    blobCounter.ObjectsOrder = ObjectsOrder.Size;
    Bitmap alterimage = new Bitmap(image);

    EuclideanColorFiltering filter = new EuclideanColorFiltering();
    filter.CenterColor = new RGB(Color.FromArgb(Red, Green, Blue));
    filter.Radius = 100;
    // apply the filter
    filter.ApplyInPlace(alterimage);

    BitmapData objectsData = alterimage.LockBits(new Rectangle(0, 0, alterimage.Width, alterimage.Height), ImageLockMode.ReadOnly, alterimage.PixelFormat);
    // grayscaling
    Grayscale grayscaleFilter = new Grayscale(0.2125, 0.7154, 0.0721);
    grayscaleFilter.Apply(new UnmanagedImage(objectsData));
    // unlock image
    alterimage.UnlockBits(objectsData);

    blobCounter.ProcessImage(alterimage);
    rects = blobCounter.GetObjectsRectangles();

    //Multi Tracking
    for (int i = 0; rects.Length > i; i++)
    {
        Rectangle objectRect = rects[i];
        using (Pen pen = new Pen(Color.FromArgb(128, 255, 0), 2))
        {
            g.DrawRectangle(pen, objectRect);
            g.DrawString((i + 1).ToString(), new Font("Arial", 12), Brushes.Red, objectRect);

        }
    }
}
```

*Figure 12: C# code for object detection*

Object detection is facilitated by utilizing a library called Aforge, which incorporates a foundational class known as blobcounter. Blobs are delineated regions identified by the algorithm, discerned from variations in light and dark intensities. The blobcounter function adeptly computes and isolates these blobs, effectively representing our objects. Subsequently, a rectangle is delineated around the blob, conforming to its width. To facilitate blob counting, a Euclidean filter is applied, facilitating the extraction of primary colors. These colors exhibit distinct intensity within the frame, facilitating differentiation between objects and the background. Further refinement is achieved by applying a grayscale filter, enhancing the blob counter's accuracy in counting by accommodating more subtle color intensities based on the grayscale comparison. The corresponding flowchart and code segment are presented in Figure 10 and Figure 11, respectively.

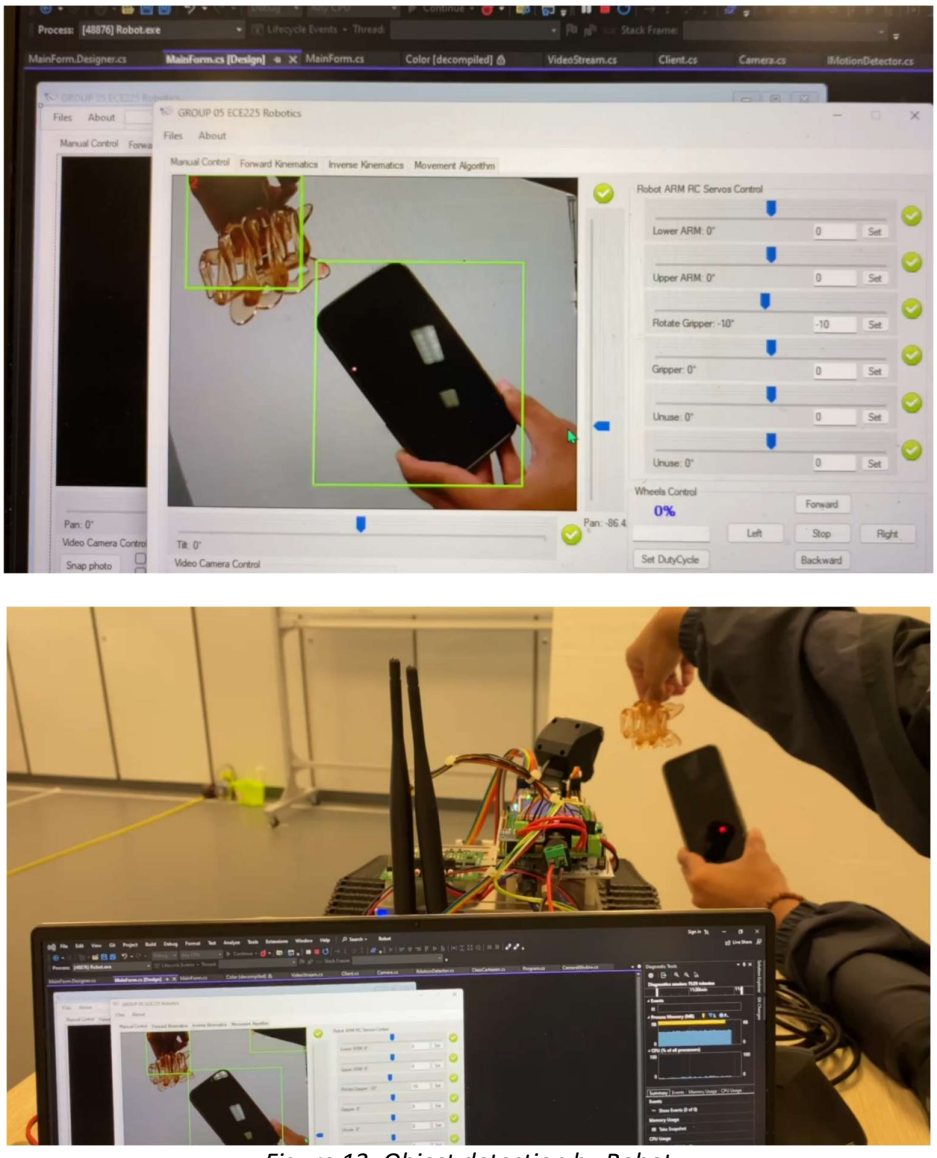



*Figure 13: Object detection by Robot*

In Figure 13 it can be seen that the robot is able to detect the two objects. The green box is drawn around the box to highlight the objects in the video stream.

### 2.2.4.2 Color Detection

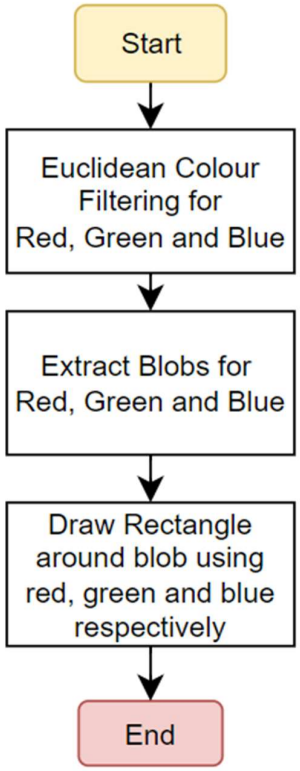
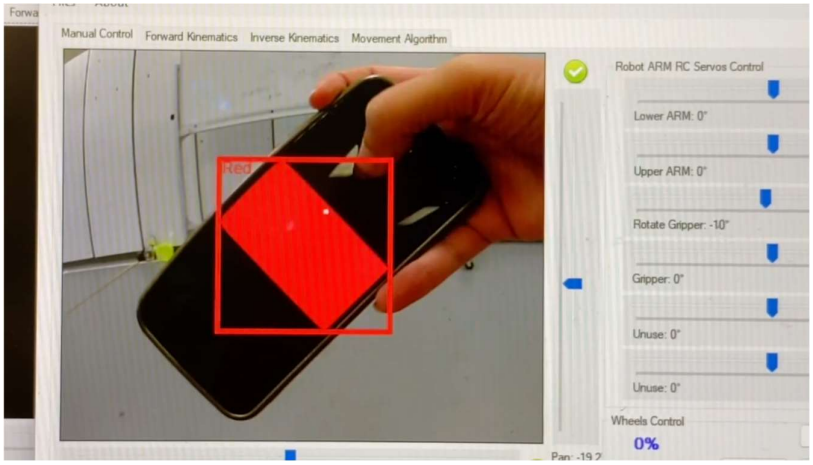


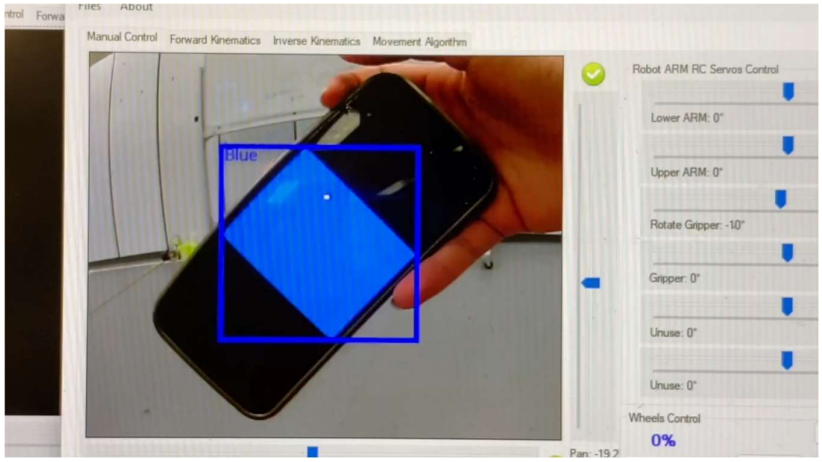

*Figure 14: Flow Chart for Color detection*

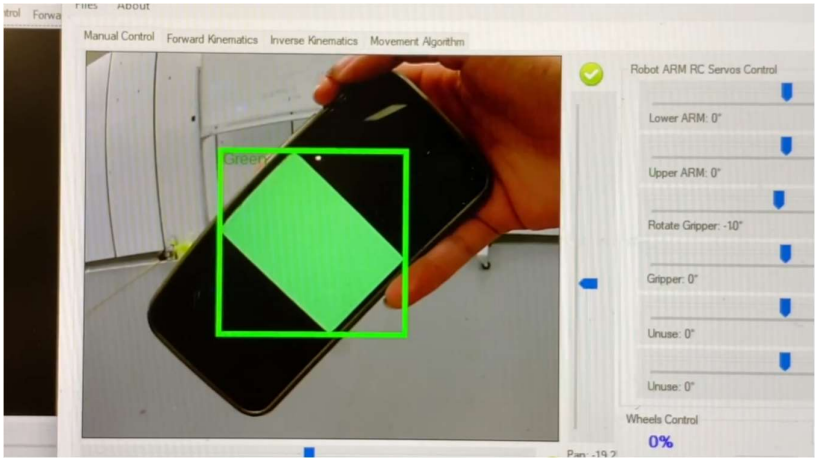
```
1 reference
private void Findcolor(Bitmap bitmap, Graphics g)
{
    Rectangle rc = this.ClientRectangle;
    // 3 bitmap objects for 3 color filters
    Bitmap imageR = new Bitmap(bitmap);
    Bitmap imageG = new Bitmap(bitmap);
    Bitmap imageB = new Bitmap(bitmap);

    BlobCounter blobCounter = new BlobCounter();
    blobCounter.MinWidth = 80;
    blobCounter.MinHeight = 80;
    blobCounter.FilterBlobs = true;

    // create filter for RED
    EuclideanColorFiltering filterR = new EuclideanColorFiltering();
    filterR.CenterColor = new RGB(Color.FromArgb(200, 5, 5));
    filterR.Radius = 100;
    Bitmap mapR = filterR.Apply(imageR);
    blobCounter.ProcessImage(mapR);
    Rectangle[] rectsR = blobCounter.GetObjectsRectangles();


    for (int i = 0; i < rectsR.Length; i++)
    {
        Rectangle objectRect = rectsR[i];
        using (Pen pen = new Pen(Color.FromArgb(255, 30, 30), 5))
        {
            g.DrawRectangle(pen, objectRect);
            g.DrawString("Red", new Font("Arial", 12), Brushes.Red, objectRect);
        }
    }
```

*Figure 15: C# code for Color detection*

The Aforge library's Euclidean filter serves the purpose of color detection, allowing for the extraction of specific colors such as red, blue, and green. This filter is applied individually for each color, with the process involving the analysis of primary RGB colors within the images. Subsequently, blobs are extracted to outline the shape of the item, followed by the drawing of a bounding box around the colored object to denote its color. Figure 14 presents the corresponding flowchart, while Figure 15 provides a section of the code segment. The code snippet illustrates the detection process for the red color, with similar procedures applied for green and blue detection.






### 2.2.4.3 Shape Detection

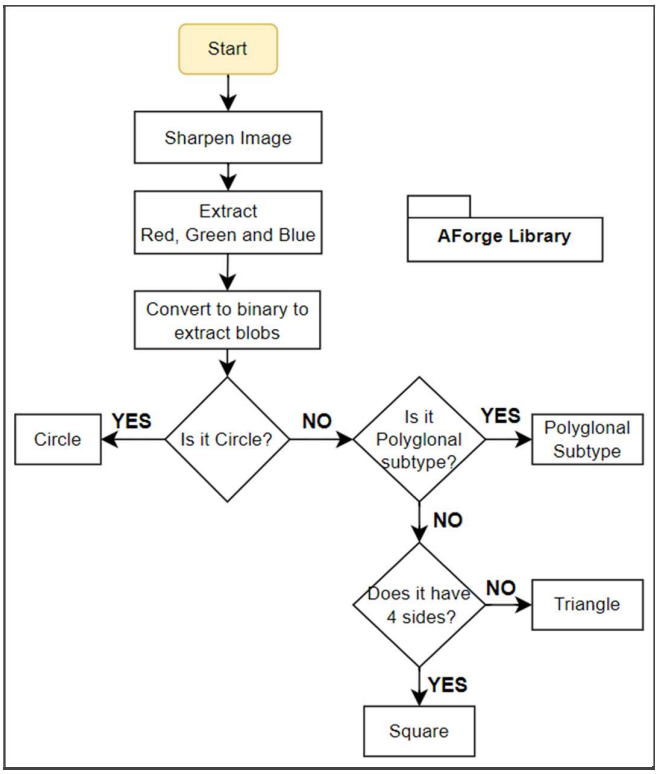



*Figure 16: Flow Chart for Shape detection*

```
// Process image
1 reference
private void FindShape(Bitmap bitmap, Graphics g)
{
    GaussianSharpen filter1 = new GaussianSharpen();
    filter1.ApplyInPlace(bitmap);
    BitmapData bitmapData = bitmap.LockBits(
        new Rectangle(0, 0, bitmap.Width, bitmap.Height),
        ImageLockMode.ReadWrite, bitmap.PixelFormat);

    // Step 1 - Set background to Black color
    ColorFiltering colorFilter = new ColorFiltering();

    // Set Filter RGB range
    colorFilter.Red = new IntRange(0, 64);
    colorFilter.Green = new IntRange(0, 64);
    colorFilter.Blue = new IntRange(0, 64);
    // Set other colors to false
    colorFilter.FillOutsideRange = false;
    // Apply RGB filter
    colorFilter.ApplyInPlace(bitmapData);

    // Step 2 - Find Objects
    BlobCounter blobCounter = new BlobCounter();

    blobCounter.FilterBlobs = true;
    blobCounter.MinHeight = 50;
    blobCounter.MinWidth = 50;

    blobCounter.ProcessImage(bitmapData);
    Blob[] blobs = blobCounter.GetObjectsInformation();

    bitmap.UnlockBits(bitmapData);
```

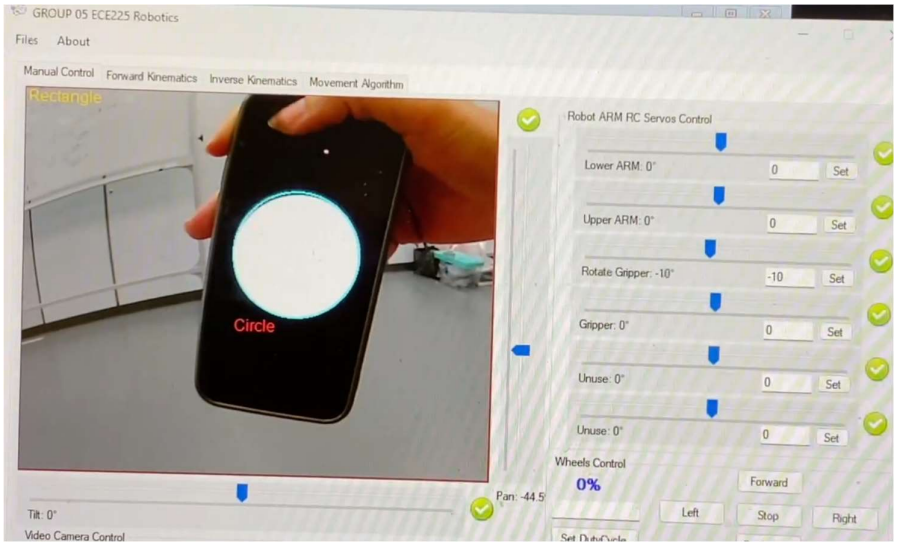
*Figure 17: C# code for Shape detection*

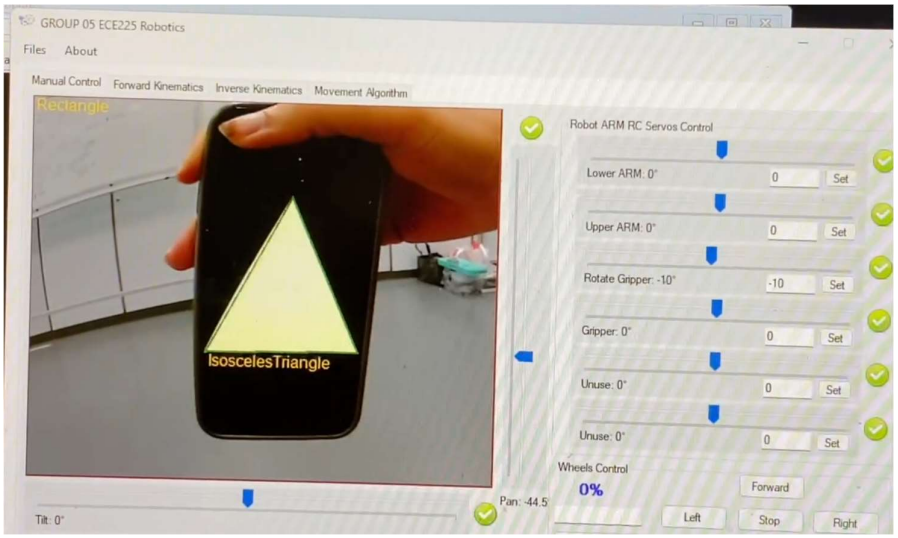
```
// Step 3 - Check shape and highlight
SimpleShapeChecker shapeChecker = new SimpleShapeChecker();

Pen yellowPen = new Pen(Color.Aqua, 2); // circles
Pen redPen = new Pen(Color.Red, 2);       // quadrilateral
Pen brownPen = new Pen(Color.Brown, 2);   // quadrilateral with known sub-type
Pen greenPen = new Pen(Color.Green, 2);   // known triangle
Pen bluePen = new Pen(Color.Blue, 2);     // triangle

for (int i = 0, n = blobs.Length; i < n; i++)
{
    List<IntPoint> edgePoints = blobCounter.GetBlobsEdgePoints(blobs[i]);

    AForge.Point center;
    float radius;

    // Check if circle
    if (shapeChecker.IsCircle(edgePoints, out center, out radius))
    {
        g.DrawEllipse(yellowPen,
            (float)(center.X - radius), (float)(center.Y - radius),
            (float)(radius * 2), (float)(radius * 2));
        g.DrawString("Circle", new Font("Arial", 12), Brushes.Red, (float)(center.X - radius), (float)(center.Y + radius));
    }
    else
    {

        List<IntPoint> corners;

        // Check if triangle or quadrilateral
        if (shapeChecker.IsConvexPolygon(edgePoints, out corners))
        {
            // get sub-type
            PolygonSubType subType = shapeChecker.CheckPolygonSubType(corners);

            Pen pen;
```

*Figure 18: C# code for Shape detection*

The simpleshapechecker class from the Aforge library serves the purpose of detecting the shape of an object. Initially, the image undergoes sharpening to enhance the clarity of edges and lines. Subsequently, an RGB filter is applied to darken the background, after which the blobs are processed. These blobs are then analyzed using the simpleshapechecker to determine their shape. In this project, the detected object is initially examined to ascertain if it is a circle; if not, it undergoes further

evaluation to determine if it matches any known polygon types within the polygon subtype class. If it doesn't match any known types, it is categorized based on the number of sides, where objects with four sides are identified as quadrilaterals, and those with fewer sides are categorized as triangles. The flowchart for this process is depicted in Figure 16, while Figures 17 and 18 provide the code segments for shape detection.



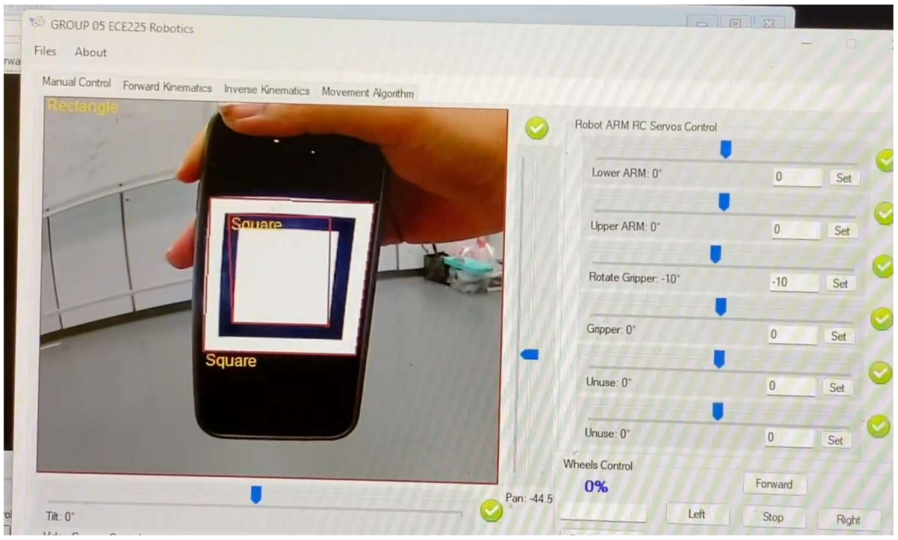



*Figure 19: Shape detection by robot*

## 2.2.4.4 Canny Edge Detection

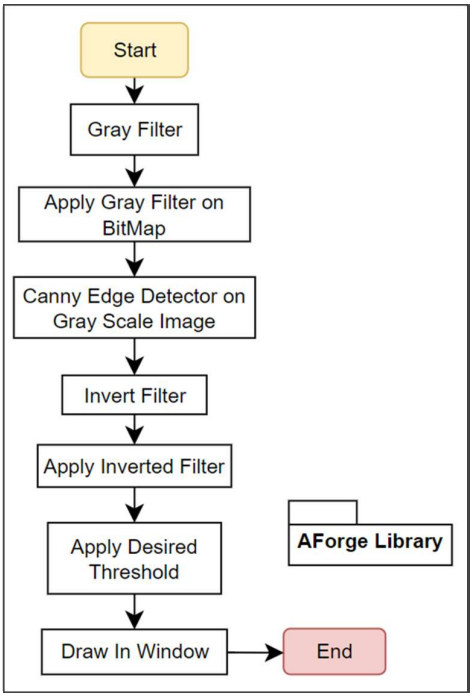



*Figure 20: Flow chart for Canny Edge Detection*

```
1 reference
private  void FindCanny (Bitmap bitmap, Graphics g)
{
    Rectangle rc = this.ClientRectangle;
    Grayscale gray = new Grayscale(0.2125, 0.7154, 0.0721); //gray scale filter
    Bitmap img = gray.Apply(bitmap);
    // Apply Canny Filter
    CannyEdgeDetector canny = new CannyEdgeDetector(0, 70);
    canny.ApplyInPlace(img);
    // Apply Inverting Filter
    Invert invrfilter = new Invert();
    invrfilter.ApplyInPlace(img);
    // Apply Threshold
    Threshold thresh = new Threshold();
    thresh.ApplyInPlace(img);
    g.DrawImage(img, rc.X + 1, rc.Y + 1, rc.Width - 2, rc.Height - 2);
}
```

*Figure 21: C# code for Canny Edge Detection*

The Canny Edge filter, sourced from the Aforge library, is used on a grayscale image to produce a Canny image, featuring distinct light edges against a dark background. To enhance clarity when viewed from our graphical user interface, the image is inverted to achieve a lighter appearance. Adjustments to the threshold value are made to eliminate unnecessary noise from the picture. The corresponding flowchart and code segment are depicted in Figure 20 and Figure 21, respectively.

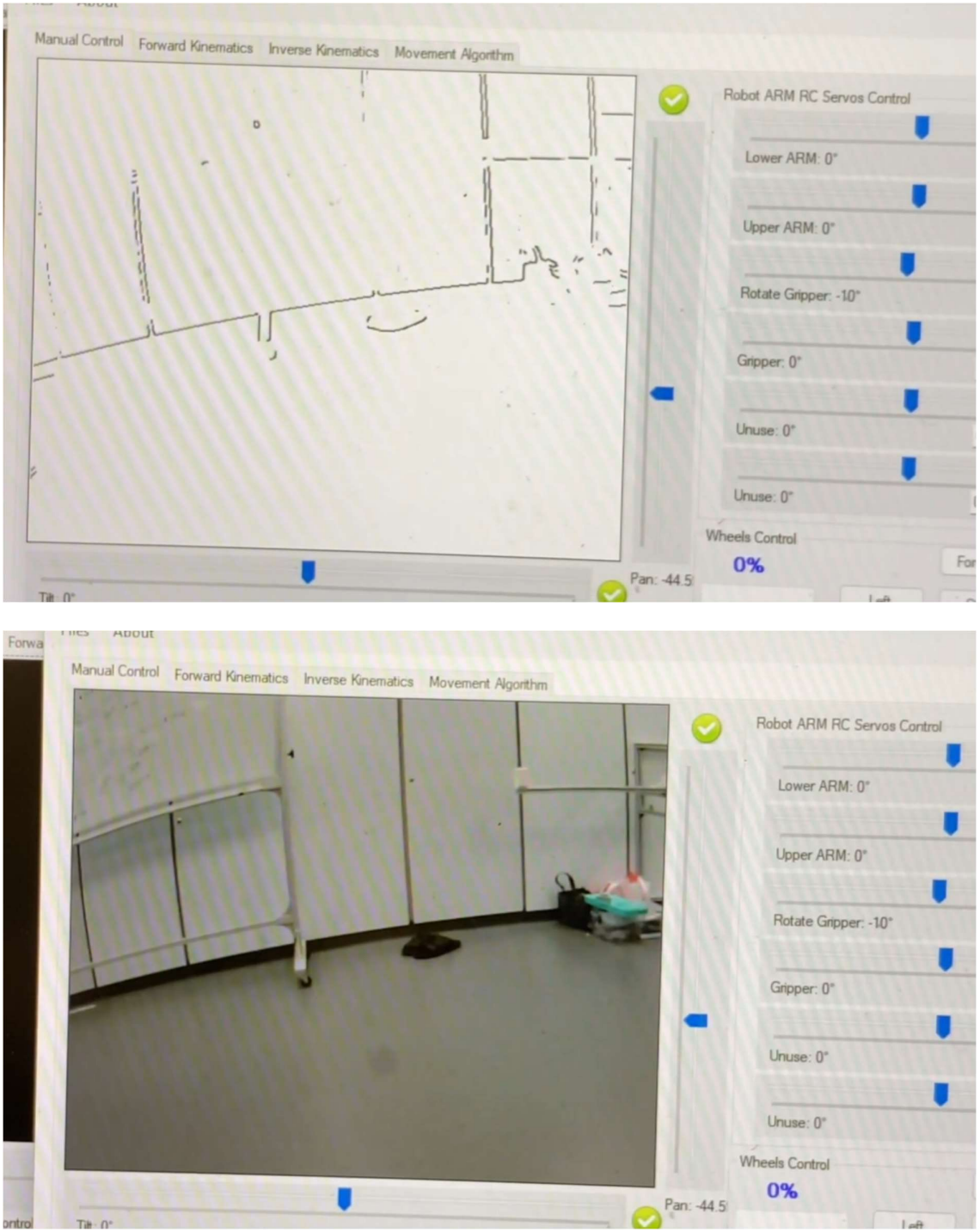



*Figure 22: Canny Edge Detection by robot*

### 2.2.4.5 Sobel Edge Detection

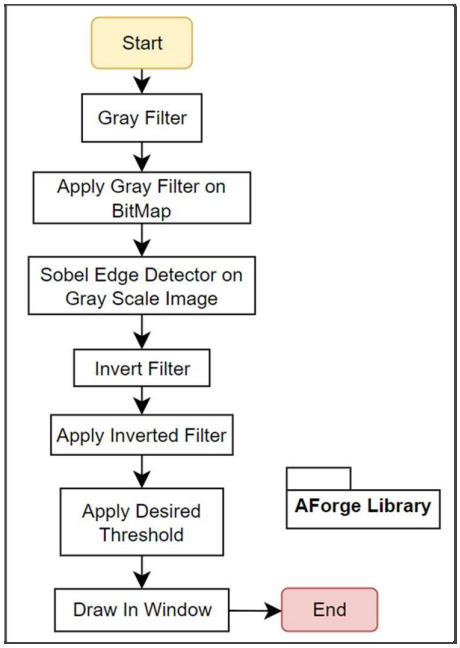



*Figure 23: Flow chart for Sobel Edge Detection*

```
1 reference
private void FindSobel(Bitmap bitmap, Graphics g)
{
    Rectangle rc = this.ClientRectangle;
    Grayscale gray = new Grayscale(0.2125, 0.7154, 0.0721); //gray scale filter
    Bitmap img = gray.Apply(bitmap);
    // Apply Sobel Filter
    SobelEdgeDetector filter = new SobelEdgeDetector();
    filter.ApplyInPlace(img);
    // Apply Inverting Filter
    Invert invrfilter = new Invert();
    invrfilter.ApplyInPlace(img);
    // Apply Threshold
    Threshold thresh = new Threshold();
    thresh.ApplyInPlace(img);

    g.DrawImage(img, rc.X + 1, rc.Y + 1, rc.Width - 2, rc.Height - 2);
}
```

*Figure 24: C# code for Sobel Edge Detection*

The Sobel Edge filter, utilized from the Aforge library, is employed on a grayscale image to generate a Sobel image featuring distinct light edges against a dark background. To enhance clarity when viewing from our graphical user interface, the image is inverted to achieve a lighter appearance. Adjustments to the threshold value are made to eliminate extraneous noise from the image. Figure 23 illustrates the corresponding flowchart, while Figure 24 presents the relevant code segment.

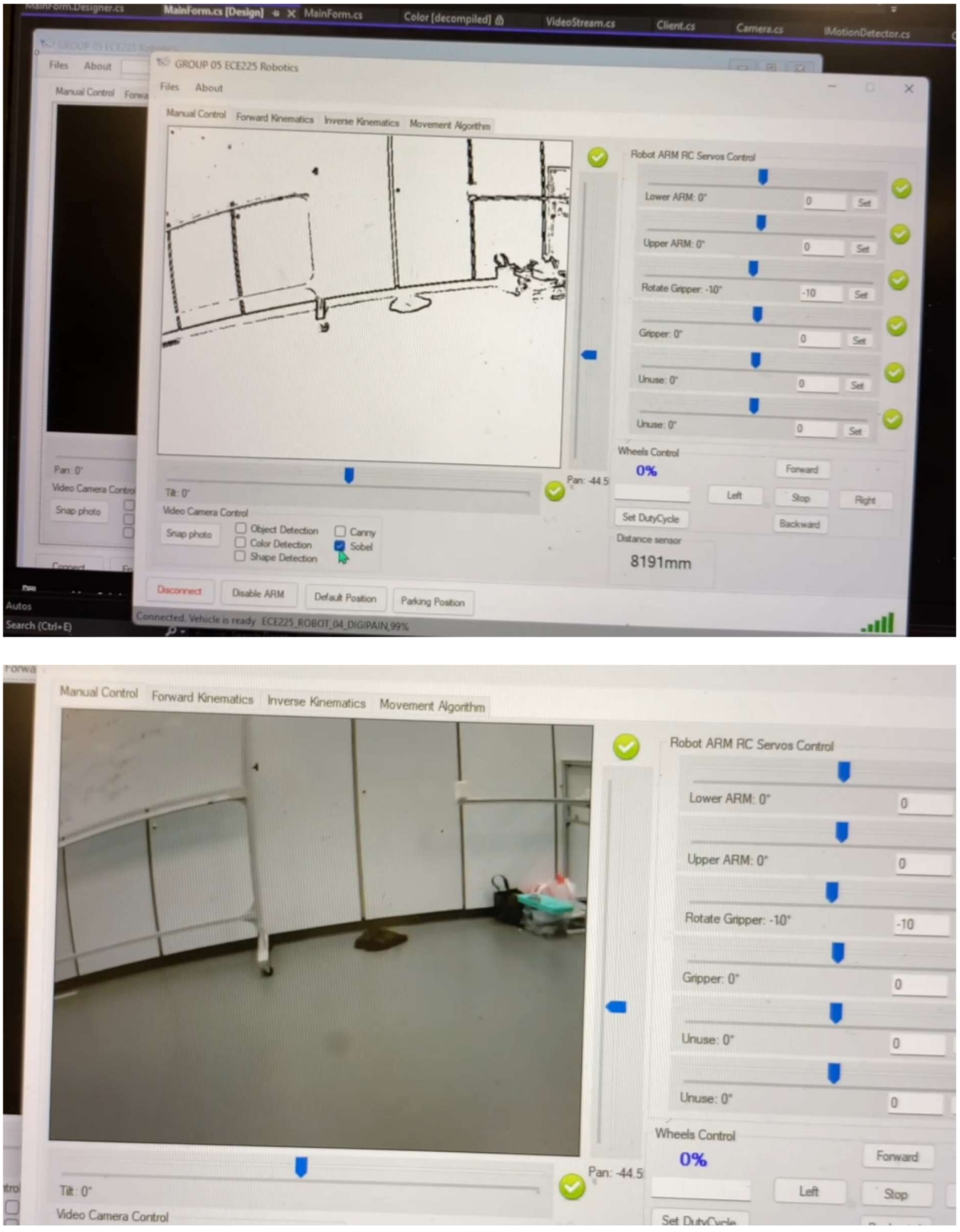



*Figure 25: Sobel Edge Detection by robot*

### 2.2.4.6 Draw square box

This feature allows the robot to use its robotic arm to sketch a square box. Upon the user clicking the "Square Draw" button, depicted in Figure 9 within the GUI section, the robotic arm initiates the process of drawing the square.

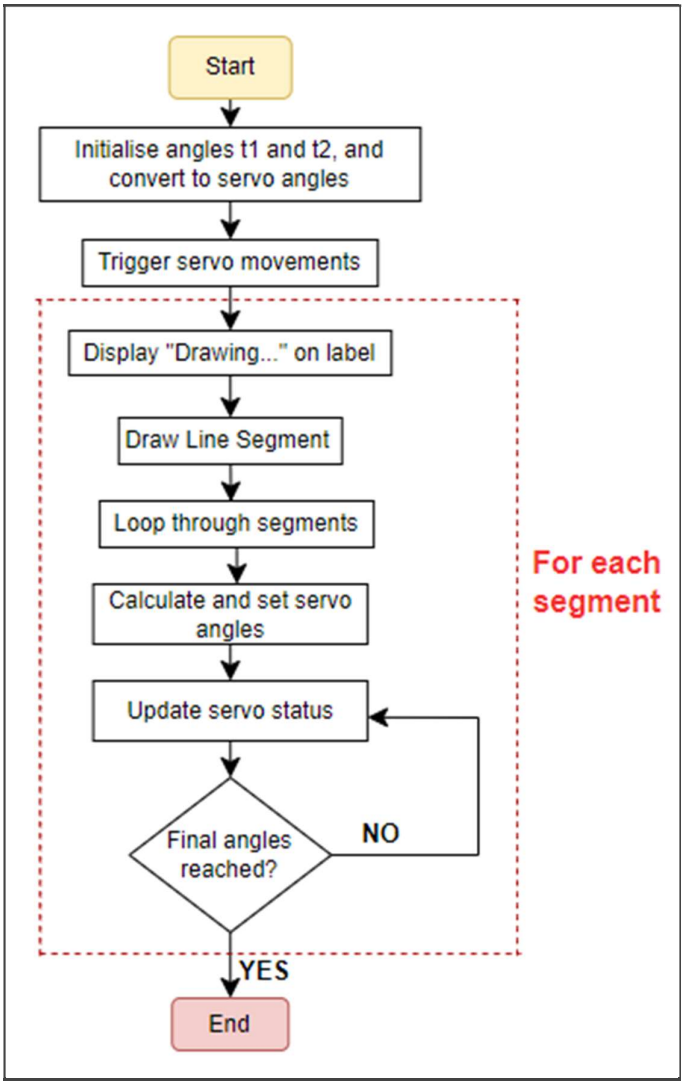



*Figure 26: Flow Chart for Draw Square*

```
1 reference
private void btnRunSquare_Click(object sender, EventArgs e)
{
    double t1 = ((-16.59145));
    double t2 = ((123.92206));

    int angle1 = Convert.ToInt32(t1 * 100.0f);
    TriggerServo(1, angle1);

    int angle2 = Convert.ToInt32(t2 * 100.0f);
    TriggerServo(2, angle2);

    Thread.Sleep(2000);
    labelFuncSts.Visible = true;
    labelFuncSts.Text = "Drawing...1";

    bool res = drawlinela(75, 125, 75, 195);
    if (res == true)
    {
        Thread.Sleep(1000);
        labelFuncSts.Visible = true;
        labelFuncSts.Text = "Drawing...2";
        bool res1 = drawlinela(75, 195, 145, 195);
        if (res1 == true)
        {
            Thread.Sleep(1000);
            labelFuncSts.Visible = true;
            labelFuncSts.Text = "Drawing...3";
            bool res2 = drawlinela(145, 195, 145, 125);
            if (res2 == true)
            {
                Thread.Sleep(1000);
                labelFuncSts.Visible = true;
                labelFuncSts.Text = "Drawing...4";
                bool res3 = drawlinela(145, 125, 75, 125);
                if (res3 == true)
                {
```

*Figure 27: C# code for Draw Square*

The function commences by receiving the current and target x and y coordinates of the robotic arm. It begins by incrementing the y coordinate of the arm by a predetermined value 'b', calculated based on the required angle for arm movement. At each step, the program verifies if the coordinate y+b has been reached. If not, the arm continues along the y-axis until it reaches this point. Once reached, the arm halts and starts decreasing its x-axis by a predefined value 'a'. Similarly, the program checks if the coordinate x-a has been reached; upon confirmation, the arm begins decreasing its y coordinate by the value 'b'. Upon reaching this point, the arm proceeds to advance along the x-axis by increasing the value by 'a'. When the coordinate x+a is attained, the arm ceases movement, completing the drawing of the square box.

### 2.2.4.7 Pick and Place of an object

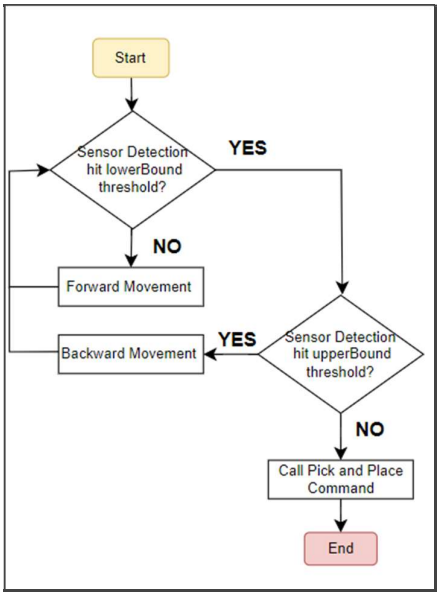



*Figure 28: Flow Chart for Square draw*

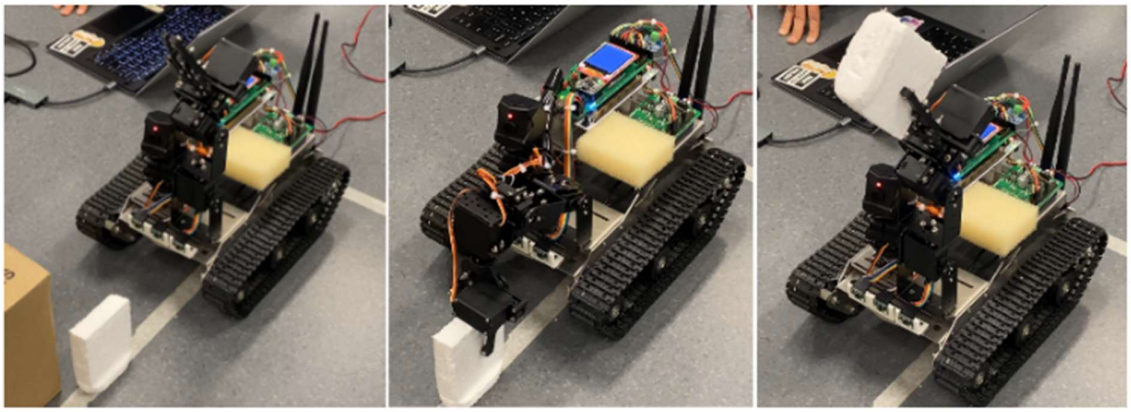


*Figure 29: Robot executing pick and place*

The pick-and-place command for an object can be activated via the host control GUI. Upon receiving the command, the robot initiates forward movement while using the laser sensor to scan for objects ahead. If the sensor detects no object within its predefined threshold, the robot continues moving forward. Upon detecting an object within the threshold range, the robot halts its movement and extends its robotic arm forward to pick up and place the object at the same location. If the detected object is below the threshold, the robot adjusts its position by moving backward.

# 4.Conclusion

## 4.1 Summary of achievements

Through this project, the team successfully translated classroom theories into practical applications. By actively applying and assimilating the learned concepts, the team gained firsthand experience in the following areas:

1. Understanding the impact of robotic arm link configuration coordinates on servo motor operations.
2. Executing fundamental image processing tasks utilizing camera technology.

## 4.2 Challenges Faced

● Robot Equations Computation - Delving into the concepts deeply to formulate equations essential for controlling the robotic arm in our project. While we were introduced to computing forward and inverse kinematic equations during lectures, adapting them to our specific robot scenario posed a slightly more intricate challenge. We needed to apply our knowledge effectively and think critically to derive these equations accurately.

● C# Programming - Familiarizing ourselves with the syntax and utilizing provided templates to incorporate new functionalities for the robot's operations. A foundational C# program was initially provided, serving as a starting point. By meticulously studying and comprehending the provided code, we endeavored to grasp how to integrate additional functions in a similar manner.

● Algorithmic Logic - Mastering the logic behind tasking the robot to perform various actions, such as picking and placing objects or drawing square shapes. We had to devise logical sequences to achieve these tasks efficiently. For instance, in square drawing, we first ensured accurate line drawing, then verified the correctness of angles obtained from inverse kinematics for use. Subsequently, we identified the four coordinates necessary for the arm's movement. Additionally, challenges arose during pick-and-place tasks, particularly concerning the robot's precision in stopping at the correct distance to grasp objects. Consequently, we iteratively adjusted sensor threshold values to enhance the robot's object-picking accuracy.

● Wiring Management - Overseeing the maintenance of hardware component wiring. Initially, during the assembly of the robot's components, errors were made in wiring. Subsequent attempts to rectify these errors proved challenging due to wires being intertwined, causing disturbances when repositioning them. Consequently, rewiring had to be conducted meticulously to avoid disrupting other components. Additionally, during robot movement, lengthy wires and cables became entangled with the robot arm, necessitating careful taping and grouping of wires to ensure unhindered movement.

## 4.3 Lessons Learnt

This project has provided us with the opportunity to apply the robotics knowledge acquired in lectures towards the development of an experimental intelligent robotic system. We particularly focused on utilizing the concepts of inverse and forward kinematics to enable movements of the 4 Degree of Freedom (DOF) manipulators. Additionally, we expanded our knowledge beyond the classroom curriculum by integrating external libraries for image processing functionalities within the smart robot system. This extension empowered the system to effectively identify, analyze, and resolve challenges associated with the implemented smart mobile manipulator. Leveraging our prior expertise in C programming, we devised algorithms to realize the envisioned robotic tasks. Consequently, the robot

can proficiently demonstrate practical functionalities such as object manipulation (pick and place), shape drawing, and color detection. Furthermore, we developed a graphical user interface using C# to facilitate remote control of the smart robot system wirelessly via a computer.