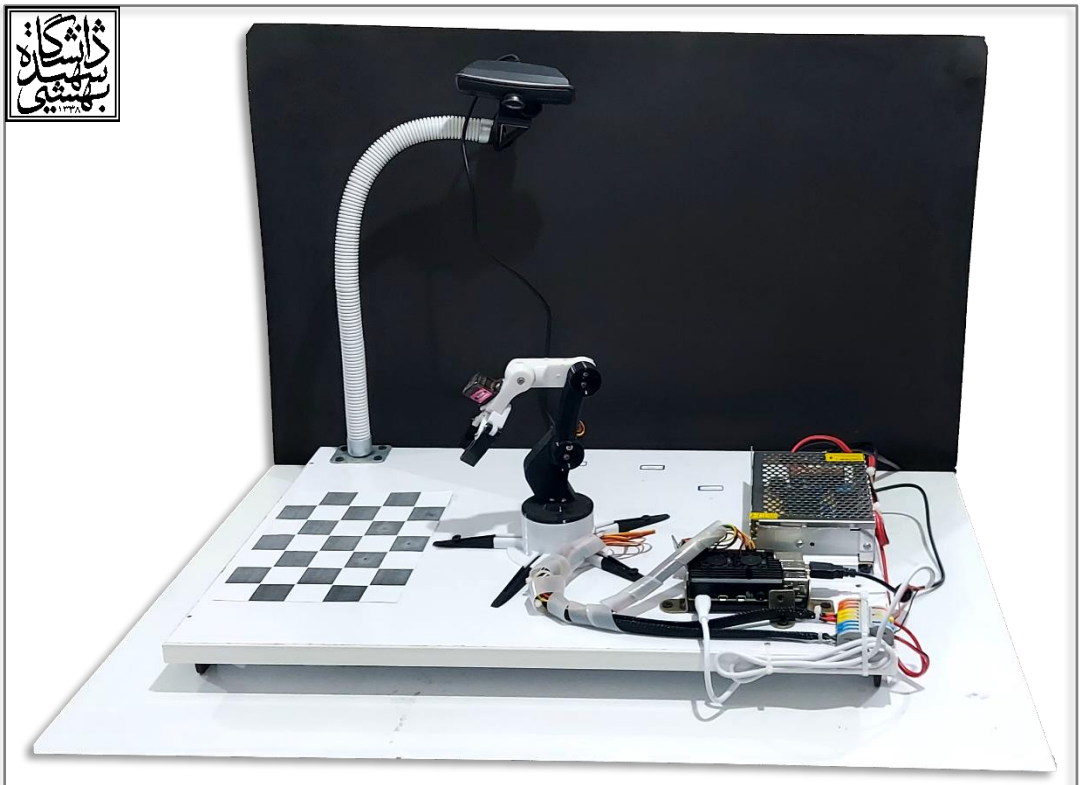


Completed on September 1, 2025

Pick–and–Place Robotic Arm

Designed and Implemented by

Mohammad Sina Majidi

Under the Supervision of

Dr. Mohsen Montazeri

Faculty of Electrical Engineering

Shahid Beheshti University

Table of Contents

Introduction

Hardware Implementation

Robot Design & Control

Software Implementation & Computer Vision

Conclusion

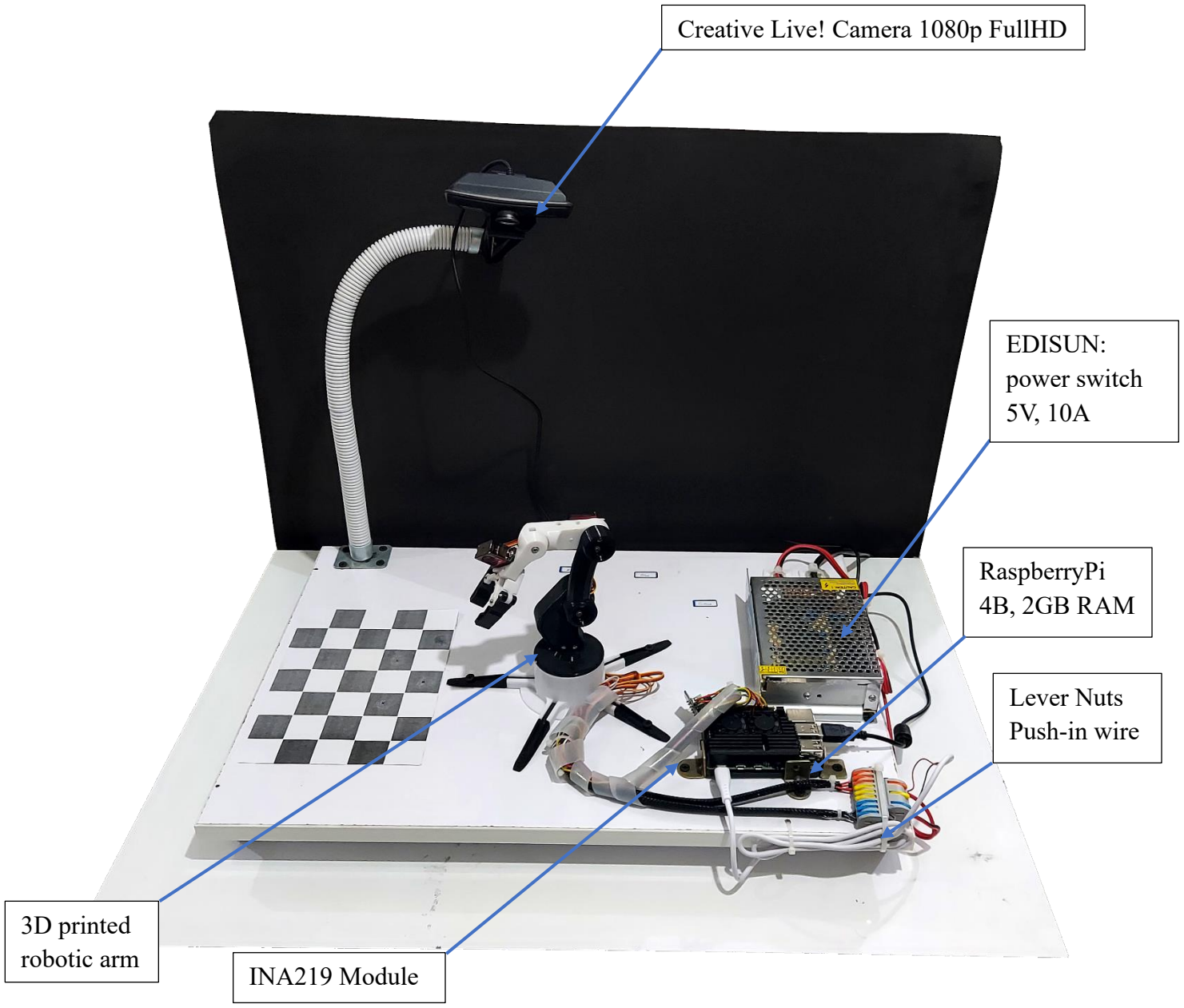
Bibliography

Introduction

The project is called the "Pick and Place Robotic Arm." My initial goal was to use a camera and object detection to pick up objects from a table and place them in a selected order based on color or geometry. I have successfully achieved my goal of simple color sorting using the pick and place method. This involves taking images with a webcam and employing computer vision techniques to identify the colors of objects located in predetermined positions. The system detects the colors and determines the x, y, and z coordinates for each object, as well as the designated locations for placing each color.

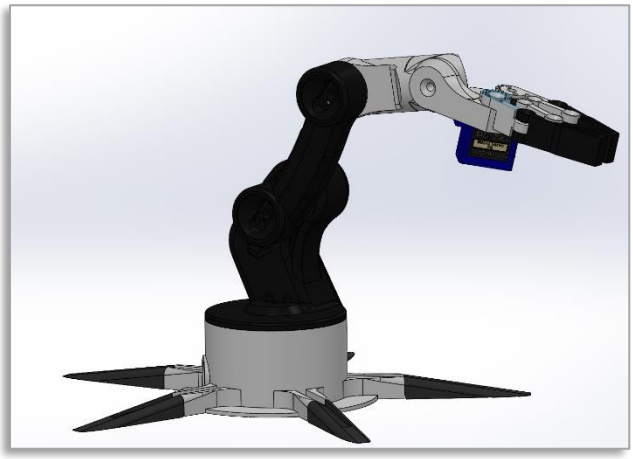
The controller uses inverse kinematics to calculate the angles of the servo motors and then execute the necessary movements. Throughout this project, I explored various methods to accomplish these tasks, which I will detail in this report. As a result, I plan to incorporate more advanced computer vision techniques in the future. Additionally, I aim for the controller to autonomously identify objects and determine their placement, building on the foundational work completed during this project.

Hardware Implementation



Robot Design & Control

The robotic arm was designed in Autodesk Fusion. I 3D printed it using ABS material with a layer thickness of 150µm. Below are the assets of the 3D model in SolidWorks and the printed parts:





Subsequently, the Denavit-Hartenberg (DH) parameters were measured using SolidWorks. Here is a brief overview of how I extracted the DH parameters[1].

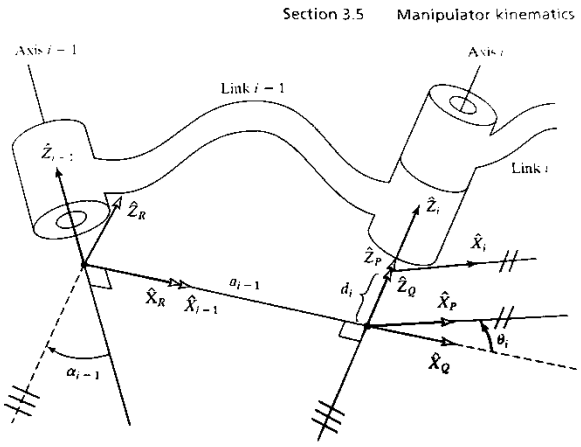
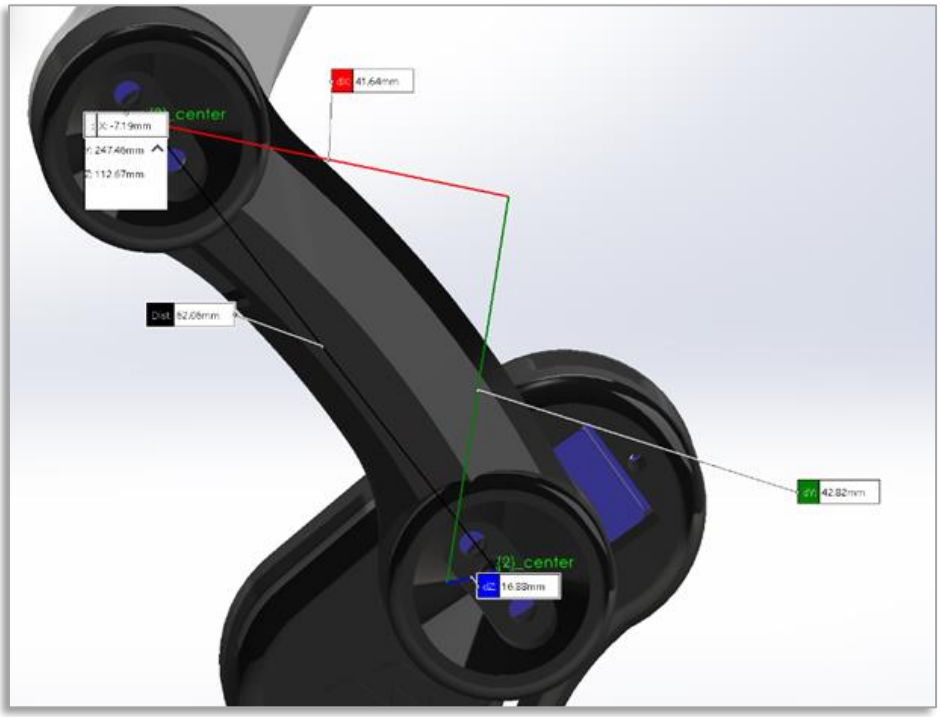
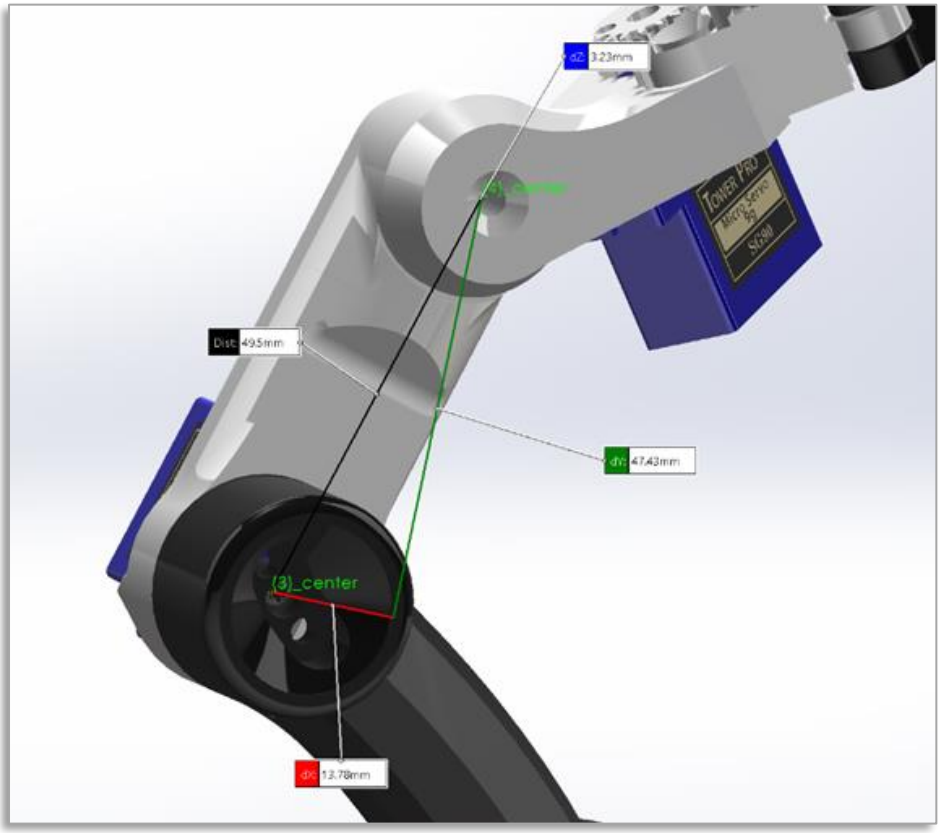
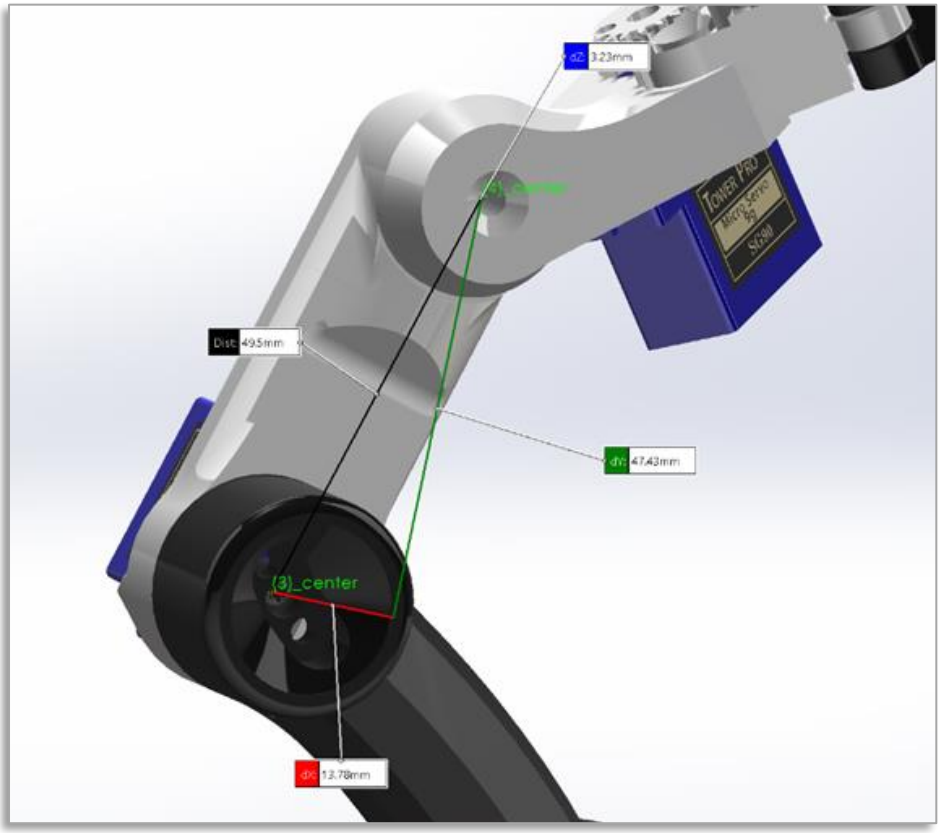


FIGURE 3.15: Location of intermediate frames $\{P\}$, $\{Q\}$, and $\{R\}$.




dZ: 3.23mm
(3)_center
Dist: 49.5mm
dY: 47.43mm
(3)_center
dX: 13.78mm


dZ: 3.23mm
(3)_center
Dist: 49.5mm
dY: 47.43mm
(3)_center
dX: 13.78mm

i	α_{i-1}	a_{i-1}	d_i	θ_i
1	0	0	0	θ_1
2	$-90°$	0	0	θ_2
3	0	62.06mm	0	θ_3
4	0	49.5mm	0	$-30°$
5	$-90°$	45.62mm	0	$6°$

DH parameters

After that, I began calculating transformation matrices based on the derived formula below:

$$ {}^{i-1}_{i}T = \begin{bmatrix} c\theta_i & -s\theta_i & 0 & a_{i-1} \\ s\theta_i c\alpha_{i-1} & c\theta_i c\alpha_{i-1} & -s\alpha_{i-1} & -s\alpha_{i-1} d_i \\ s\theta_i s\alpha_{i-1} & c\theta_i s\alpha_{i-1} & c\alpha_{i-1} & c\alpha_{i-1} d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

Where $S\theta_i = Sin(\theta_i)$ and $c\alpha_i = cos(\alpha_i)$.

MATLAB code for calculating transformation matrices using the general form and Denavit-Hartenberg (DH) parameters:

```
 syms thetai ai_1 alphai_1 di T(alphai_1,ai_1,di,thetai) theta1 theta2 theta3


 T(alphai_1,ai_1,di,thetai) = [

 cosd(thetai), -1*sind(thetai), 0, ai_1;
 sind(thetai)*cosd(alphai_1), cosd(thetai)*cosd(alphai_1), -1*sind(alphai_1), -
1*sind(alphai_1)*di;
 sind(thetai)*sind(alphai_1), cosd(thetai)*sind(alphai_1), cosd(alphai_1),
cosd(alphai_1)*di;
 0, 0, 0, 1

 ];


 % unit is mm.

 DH_params = [

 0, 0, 0, theta1;
 -90, 0, 0, theta2;
 0, 62.06, 0, theta3;
 0, 49.5, 0, -30;
 -90, 45.62, 0, 6


 ];
```

```
T01 = T(DH_params(1,1),DH_params(1,2),DH_params(1,3),DH_params(1,4))
```

$$
{}^{0}_{1}T = \begin{pmatrix} \cos\left(\frac{\pi\,\theta_1}{180}\right) & -\sin\left(\frac{\pi\,\theta_1}{180}\right) & 0 & 0 \\ \sin\left(\frac{\pi\,\theta_1}{180}\right) & \cos\left(\frac{\pi\,\theta_1}{180}\right) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
$$

```
T12 = T(DH_params(2,1),DH_params(2,2),DH_params(2,3),DH_params(2,4))
```

$$
{}^{1}_{2}T = \begin{pmatrix} \cos\left(\frac{\pi\,\theta_2}{180}\right) & -\sin\left(\frac{\pi\,\theta_2}{180}\right) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\sin\left(\frac{\pi\,\theta_2}{180}\right) & -\cos\left(\frac{\pi\,\theta_2}{180}\right) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
$$

```
T23 = T(DH_params(3,1),DH_params(3,2),DH_params(3,3),DH_params(3,4))
```

$$
{}^{2}_{3}T = \begin{pmatrix} \cos\left(\frac{\pi\,\theta_3}{180}\right) & -\sin\left(\frac{\pi\,\theta_3}{180}\right) & 0 & \frac{3103}{50} \\ \sin\left(\frac{\pi\,\theta_3}{180}\right) & \cos\left(\frac{\pi\,\theta_3}{180}\right) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
$$

```
T34 = T(DH_params(4,1),DH_params(4,2),DH_params(4,3),DH_params(4,4))
```

$$
{}^{3}_{4}T = \begin{pmatrix} \frac{\sqrt{3}}{2} & \frac{1}{2} & 0 & \frac{99}{2} \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
$$

```
T45 = T(DH_params(5,1),DH_params(5,2),DH_params(5,3),DH_params(5,4))
```

$$
{}^{4}_{5}T = \begin{pmatrix} \cos\left(\frac{\pi}{30}\right) & -\sin\left(\frac{\pi}{30}\right) & 0 & \frac{2281}{50} \\ 0 & 0 & 1 & 0 \\ -\sin\left(\frac{\pi}{30}\right) & -\cos\left(\frac{\pi}{30}\right) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
$$

At this point, I had gathered enough data to calculate the inverse kinematic solution, utilizing a numerical method based on the Newton-Raphson method and the Jacobian matrix.

To achieve the desired positions θ_i (i = 1, 2, 3) for the coordinates (x, y, z), I had to consider the following conditions related to the physical limitations of the angles:

$$\forall\, \theta_i\,, i \in \{1,2,3\}: \quad 0 \leq \theta_i \leq 180°$$

$$f_1\left(x_1^{(i+1)}, x_2^{(i+1)}, \cdots, x_n^{(i+1)}\right) \approx f_1\left(x_1^{(i)}, x_2^{(i)}, \cdots, x_n^{(i)}\right) + \frac{\partial f_1}{\partial x_1}\bigg|_{x^{(i)}}\left(x_1^{(i+1)} - x_1^{(i)}\right) + \frac{\partial f_1}{\partial x_2}\bigg|_{x^{(i)}}\left(x_2^{(i+1)} - x_2^{(i)}\right) + \cdots + \frac{\partial f_1}{\partial x_n}\bigg|_{x^{(i)}}\left(x_n^{(i+1)} - x_n^{(i)}\right)$$

$$\begin{pmatrix} f_1\left(x^{(i+1)}\right) \\ f_2\left(x^{(i+1)}\right) \\ \vdots \\ f_n\left(x^{(i+1)}\right) \end{pmatrix} = \begin{pmatrix} f_1\left(x^{(i)}\right) \\ f_2\left(x^{(i)}\right) \\ \vdots \\ f_n\left(x^{(i)}\right) \end{pmatrix} + \begin{pmatrix} \frac{\partial f_1}{\partial x_1}\big|_{x^{(i)}} & \frac{\partial f_1}{\partial x_2}\big|_{x^{(i)}} & \cdots & \frac{\partial f_1}{\partial x_n}\big|_{x^{(i)}} \\ \frac{\partial f_2}{\partial x_1}\big|_{x^{(i)}} & \frac{\partial f_2}{\partial x_2}\big|_{x^{(i)}} & \cdots & \frac{\partial f_2}{\partial x_n}\big|_{x^{(i)}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial f_n}{\partial x_1}\big|_{x^{(i)}} & \frac{\partial f_n}{\partial x_2}\big|_{x^{(i)}} & \cdots & \frac{\partial f_n}{\partial x_n}\big|_{x^{(i)}} \end{pmatrix} \begin{pmatrix} \left(x_1^{(i+1)} - x_1^{(i)}\right) \\ \left(x_2^{(i+1)} - x_2^{(i)}\right) \\ \vdots \\ \left(x_n^{(i+1)} - x_n^{(i)}\right) \end{pmatrix}$$

By setting the left-hand side to zero, we obtain:

$$\begin{pmatrix} \frac{\partial f_1}{\partial x_1}\Big|_{x^{(i)}} & \frac{\partial f_1}{\partial x_2}\Big|_{x^{(i)}} & \cdots & \frac{\partial f_1}{\partial x_n}\Big|_{x^{(i)}} \\ \frac{\partial f_2}{\partial x_1}\Big|_{x^{(i)}} & \frac{\partial f_2}{\partial x_2}\Big|_{x^{(i)}} & \cdots & \frac{\partial f_2}{\partial x_n}\Big|_{x^{(i)}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial f_n}{\partial x_1}\Big|_{x^{(i)}} & \frac{\partial f_n}{\partial x_2}\Big|_{x^{(i)}} & \cdots & \frac{\partial f_n}{\partial x_n}\Big|_{x^{(i)}} \end{pmatrix} \begin{pmatrix} \left(x_1^{(i+1)} - x_1^{(i)}\right) \\ \left(x_2^{(i+1)} - x_2^{(i)}\right) \\ \vdots \\ \left(x_n^{(i+1)} - x_n^{(i)}\right) \end{pmatrix} = - \begin{pmatrix} f_1\left(x^{(i)}\right) \\ f_2\left(x^{(i)}\right) \\ \vdots \\ f_n\left(x^{(i)}\right) \end{pmatrix}$$

This can be expressed as:

$$K\Delta x = -f$$

Iteration strategy:

$$\Delta x = -K^{-1} f \Rightarrow x^{(i+1)} = x^{(i)} + \Delta x$$

Now it was the right time to calculate Jacobian matrix using MATLAB.

Jacobian Matrix:

$$\begin{pmatrix} -\frac{\pi\,\sigma_2\,\sigma_4}{18000} & -\frac{\pi\,\sigma_3\,\sigma_5}{9000} & -\frac{\pi\,\sigma_3\,\sigma_8\,\sigma_9}{9000} \\ \frac{\pi\,\sigma_3\,\sigma_4}{18000} & -\frac{\pi\,\sigma_2\,\sigma_5}{9000} & -\frac{\pi\,\sigma_2\,\sigma_8\,\sigma_9}{9000} \\ 0 & -\frac{3103\,\pi\,\sigma_7}{9000} - \frac{2281\,\pi\,\sigma_1\,\sqrt{\frac{9900\,\sqrt{3}}{2281}+\frac{45314344}{5202961}}}{18000} & -\frac{\pi\,\sigma_1\,\sigma_9}{9000} \end{pmatrix}$$

where

$$\sigma_1 = \sin\left(\operatorname{atan}\left(\sqrt{3}+\frac{4950}{2281}\right)+\frac{\pi\,\theta_2}{180}+\frac{\pi\,\theta_3}{180}\right)$$

$$\sigma_2 = \sin\left(\frac{\pi\,\theta_1}{180}\right)$$

$$\sigma_3 = \cos\left(\frac{\pi\,\theta_1}{180}\right)$$

$$\sigma_4 = 4950\cos(\sigma_6) + 2281\sin(\sigma_6) + 6206\,\sigma_7 + 2281\,\sqrt{3}\,\cos(\sigma_6)$$

$$\sigma_5 = 3103\sin\left(\frac{\pi\,\theta_2}{180}\right) + \sigma_8\,\sigma_9$$

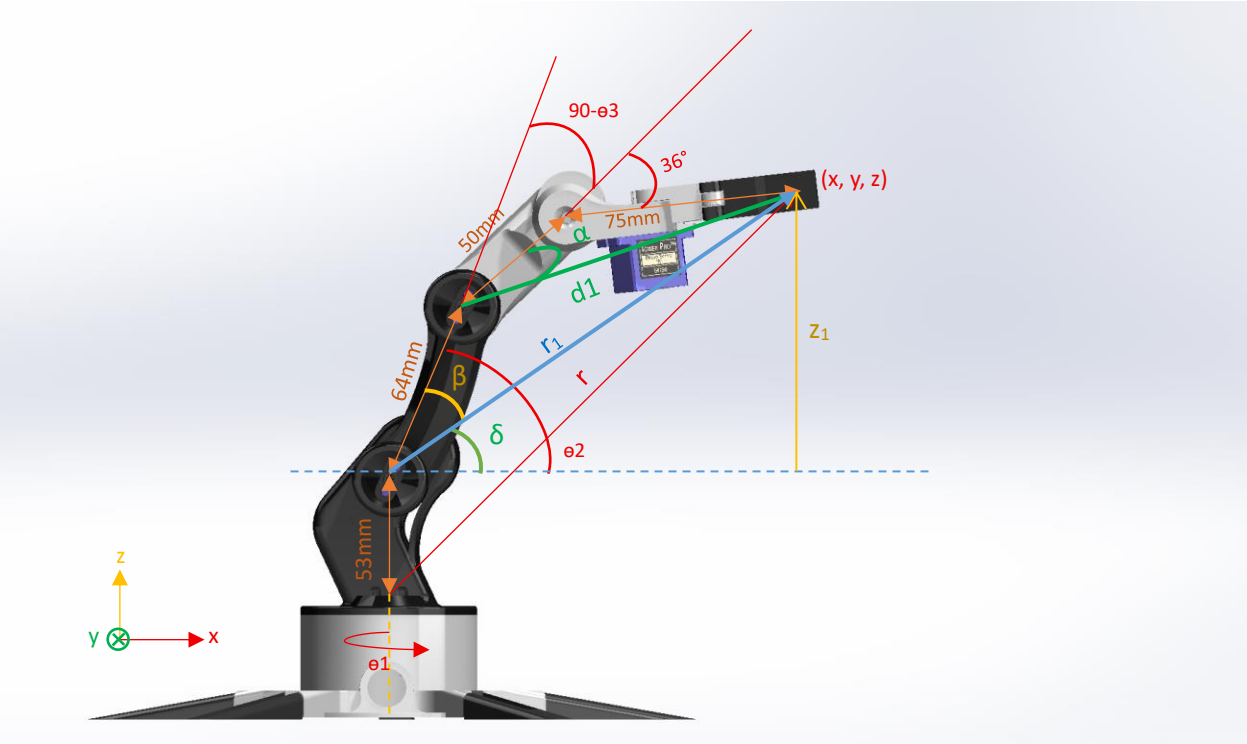
$$\sigma_6 = \frac{\pi\,(\theta_2+\theta_3)}{180}$$

$$\sigma_7 = \cos\left(\frac{\pi\,\theta_2}{180}\right)$$

$$\sigma_8 = \sin\left(\operatorname{atan}\left(\frac{5202961\,\sqrt{3}}{8893617}-\frac{3763650}{2964539}\right)+\frac{\pi\,\theta_2}{180}+\frac{\pi\,\theta_3}{180}\right)$$

$$\sigma_9 = \sqrt{5645475\,\sqrt{3}+11328586}$$

After implementing the inverse kinematics, I noticed that convergence could not be achieved at certain points within the accessible space for the end effector. As a result, the outcomes were unsatisfactory. Therefore, I decided to explore a new approach to solving inverse kinematics using a **closed-form geometric solution**:



$L_1 = 53mm$

$L_2 = 64mm$

$L_3 = 50mm$

$L_4 = 75mm$

$\phi = 36°$

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

$$d_1 = \sqrt{L_3^2 + L_4^2 - 2L_3L_4\cos(180^0 - \varphi)}$$

$$\alpha = \cos^{-1}\left(\frac{d_1^2 + L_3^2 - L_4^2}{2d_1L_3}\right)$$

$$r_1 = \sqrt{x^2 + y^2 + (z - L_1)^2}$$

$$\theta_3 = 90 - (180 - \alpha - \cos^{-1}\left(\frac{L_2^2 + d_1^2 - r_1^2}{2L_2d_1}\right))$$

$$\theta_1 = \tan^{-1}\left(\frac{y}{x}\right)$$

$$\theta_2 = \delta + \beta$$

$$z_1 = z - L_I$$

$$\delta = \cos^{-1}\left(\frac{r_1^2 + x^2 - z_1^2}{2r_1x}\right)$$

$$\beta = \cos^{-1}\left(\frac{L_2^2 + r_1^2 - d_1^2}{2L_2r_1}\right)$$

Mapping theta(s) into actuator space:

$\theta_{1(Act.)} = \theta_1 + 12°$

$\theta_{2(Act.)} = \theta_2 + 12°$

$\theta_{3(Act.)} = 90 - \theta_3 + 12°$

After making some simplifications and minor modifications, here is the implemented code along with the plotted results in MATLAB:

```
%clc; clear; close all;


%% Link lengths and derived parameter
L1    = 53;
L2    = 64;
L3    = 50;
L4    = 75;
% change 36 to 48.00 its true.
phi   = 180 - 48;
L3_p  = sqrt(L3^2 + L4^2 - 2*L3*L4*cosd(phi));


%% Grid of target points
n  = 40;
n_z = 5;
xs = linspace(-100,100,n);
ys = linspace(-100,100,n);
zs = linspace(-40,100,n_z);
[XX,YY,ZZ] = ndgrid(xs, ys, zs);
pts = numel(XX);


%% Preallocate
BASE = nan(pts,1);
ELB  = nan(pts,1);
SHO  = nan(pts,1);


%% Flattened point list
```

```
X = XX(:);  Y = YY(:);  Z = ZZ(:);


%% Compute IK
for i = 1:pts
    [b,e,s] = IK_point(X(i), Y(i), Z(i), L1, L2, L3_p);
    BASE(i) = b;
    ELB(i)  = e;
    SHO(i)  = s;
end


%% Plot setup
figure('Position',[100 100 1200 400]);
angleSets = {BASE, ELB, SHO};
titles    = {'Base Angle (deg)', 'Elbow Angle (deg)', 'Shoulder Angle (deg)'};


for ax = 1:3
    angles = angleSets{ax};

    % Classify
    isCpx    = imag(angles)~=0;
    isReal   = ~isCpx;
    isOOR    = isReal & ( angles<0 | angles>180 );
    isValid  = isReal & ~isOOR;

    % Normalize valid angles into [0,1] for colormap
    cmap = jet(256);
    vals = angles(isValid);
    idxC = round( (vals/180) * 255 ) + 1;
    idxC = min(max(idxC,1),256);
    cmapsel = cmap(idxC, :);


    subplot(1,3,ax)
    hold on

    % 1) valid, colored by angle
    scatter3( X(isValid), Y(isValid), Z(isValid), 15, cmapsel, 'filled');

    % 2) out-of-range in red
    scatter3( X(isOOR), Y(isOOR), Z(isOOR), 15, [1 0 0], 'filled' );

    % 3) complex in black
    scatter3( X(isCpx), Y(isCpx), Z(isCpx), 15, [0 0 0], 'filled' );
```

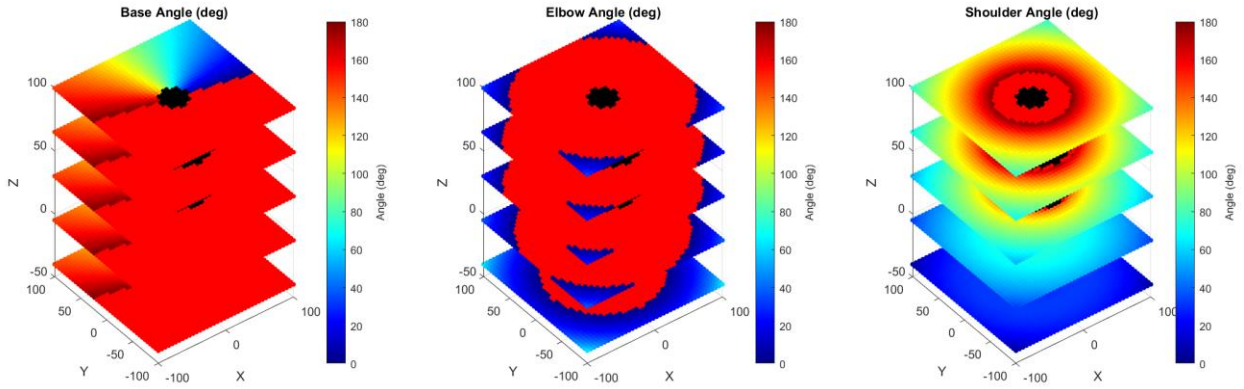
```
    title(titles{ax})
    xlabel('X'); ylabel('Y'); zlabel('Z')
    colormap(jet)
    c = colorbar;
    c.Label.String = 'Angle (deg)';
    caxis([0 180])
    view(3)
    grid on
end


%% IK helper function
function [base, elbow, shoulder] = IK_point(x, y, z, L1, L2, L3_p)
    % Shift by base height
    z_p = z - L1;
    r   = sqrt(x^2 + y^2 + z_p^2);

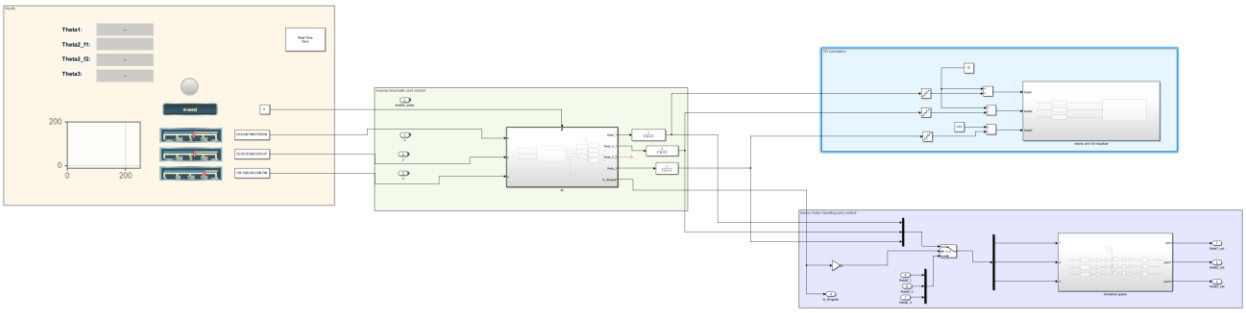
    % Check reachability
    if r > (L2 + L3_p) || r < abs(L2 - L3_p)
        base    = NaN + 1i;
        elbow   = NaN + 1i;
        shoulder= NaN + 1i;
        return
    end


    % Base rotation
    base = atan2d(y, x);


    % Elbow via law of cosines
    cos_elb = (r^2 - L2^2 - L3_p^2) / (2 * L2 * L3_p);
    cos_elb = min(max(cos_elb, -1), +1);
    elbow   = acosd(cos_elb);
    % Shoulder elevation
    alpha    = atan2d(z_p, sqrt(x^2 + y^2));
    beta     = acosd((r^2 + L2^2 - L3_p^2) / (2 * L2 * r));
    shoulder = alpha + beta;
    % space mapper:
    base = base +12;
    shoulder = shoulder +12;
    elbow = 102-elbow;
end
```

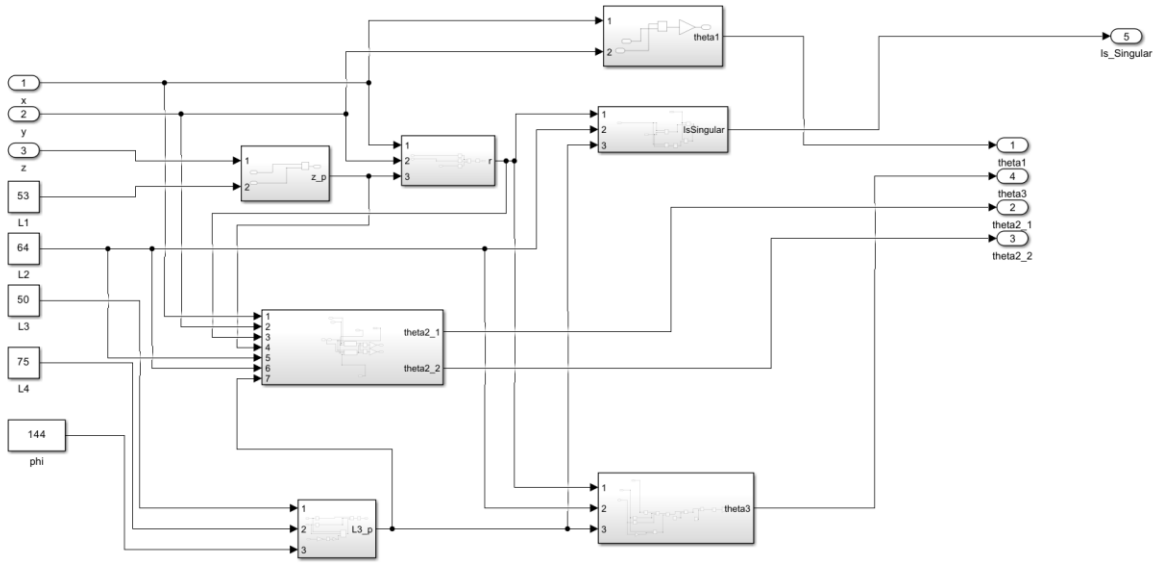


I implemented it in Simulink using blocks that can simulate and upload to the ESP32 to change the angles of servos.



I have utilized and configured the blocks in fixed-point format, allowing HDL Coder to generate Verilog (or VHDL) code easily for upload to Xilinx FPGA series. Implementing this on an FPGA is planned for the near future.

Within the inverse kinematics block, the following terminology applies:



After implementing the inverse kinematics on the ESP32, the results were unsatisfactory and seemed invalid in certain ranges. I had previously solved the inverse kinematics using closed-form geometric solutions multiple times, but the results were not accurate and were invalid in some ranges. Unable to identify the cause of these issues, I decided to switch to a more autonomous numerical inverse kinematics solver.

I utilized the IKPy[2] library, which employs an optimization-based iterative numerical solver. This approach addresses the inverse kinematics problem by formulating it as a nonlinear least-squares problem with variable bounds, using the Levenberg–Marquardt algorithm.

Here is how the Levenberg–Marquardt algorithm works:

{The text below is part of reference 3. Please refer there for the complete text.}

The Levenberg-Marquardt algorithm adaptively varies the coefficient updates between the gradient descent update and the Gauss-Newton update,

$$\left[\boldsymbol{J}^{\mathsf{T}}\boldsymbol{W}\boldsymbol{J} + \lambda\boldsymbol{I}\right]\boldsymbol{h}_{\mathsf{lm}} = \boldsymbol{J}^{\mathsf{T}}\boldsymbol{W}(\boldsymbol{y} - \hat{\boldsymbol{y}}) \ , \qquad (12)$$

where small values of the damping coefficient λ result in a Gauss-Newton update and large values of λ result in a gradient descent update. The damping coefficient λ is initialized to be large so that first updates are small steps in the steepest-descent direction. If any iteration happens to result in a worse approximation (χ2(a + hlm) > χ2(a)), then λ is increased. Otherwise, as the solution improves, λ is decreased, the Levenberg-Marquardt method approaches the Gauss-Newton method, and the solution typically accelerates to the local minimum [8, 9, 10].

In Marquardt's update relationship [10], the damping coefficient λ is scaled by the diagonal of the Hessian JTWJ for each coefficient.

$$\left[\boldsymbol{J}^{\mathsf{T}}\boldsymbol{W}\boldsymbol{J} + \lambda\ \mathsf{diag}(\boldsymbol{J}^{\mathsf{T}}\boldsymbol{W}\boldsymbol{J})\right]\boldsymbol{h}_{\mathsf{lm}} = \boldsymbol{J}^{\mathsf{T}}\boldsymbol{W}(\boldsymbol{y} - \hat{\boldsymbol{y}}) \ , \qquad (13)$$

Numerical Implementation (13) Many variations of the Levenberg-Marquardt have been published in papers and in code, e.g., [6, 8, 11, 12, 13]. This document borrows from some of these. In iteration i, the step h is evaluated by comparing χ2(a) to χ2(a +h). The step is accepted if the metric ρi [11] is greater than a user-specified threshold, ϵ4 > 0. This metric is a measure of the actual improvement in χ2 as compared to the improvement of an LM update assuming the approximation (8) were exact.

$$\rho_i(\boldsymbol{h}_{\text{lm}}) = \frac{\chi^2(\boldsymbol{a}) - \chi^2(\boldsymbol{a}+\boldsymbol{h}_{\text{lm}})}{|\,(\boldsymbol{y}-\hat{\boldsymbol{y}})^\mathsf{T}\boldsymbol{W}(\boldsymbol{y}-\hat{\boldsymbol{y}}) - (\boldsymbol{y}-\hat{\boldsymbol{y}}-\boldsymbol{J}\boldsymbol{h}_{\text{lm}})^\mathsf{T}\boldsymbol{W}(\boldsymbol{y}-\hat{\boldsymbol{y}}-\boldsymbol{J}\boldsymbol{h}_{\text{lm}})\,|} \tag{14}$$

$$= \frac{\chi^2(\boldsymbol{a}) - \chi^2(\boldsymbol{a}+\boldsymbol{h}_{\text{lm}})}{|\,\boldsymbol{h}_{\text{lm}}^\mathsf{T}\,(\lambda_i\boldsymbol{h}_{\text{lm}} + \boldsymbol{J}^\mathsf{T}\boldsymbol{W}\,(\boldsymbol{y}-\hat{\boldsymbol{y}}(\boldsymbol{a})))\,|} \quad \text{if using eq'n (12) for } \boldsymbol{h}_{\text{lm}} \tag{15}$$

$$= \frac{\chi^2(\boldsymbol{a}) - \chi^2(\boldsymbol{a}+\boldsymbol{h}_{\text{lm}})}{|\,\boldsymbol{h}_{\text{lm}}^\mathsf{T}\,(\lambda_i\text{diag}(\boldsymbol{J}^\mathsf{T}\boldsymbol{W}\boldsymbol{J})\boldsymbol{h}_{\text{lm}} + \boldsymbol{J}^\mathsf{T}\boldsymbol{W}\,(\boldsymbol{y}-\hat{\boldsymbol{y}}(\boldsymbol{a})))\,|} \quad \text{if using eq'n (13) for } \boldsymbol{h}_{\text{lm}} \tag{16}$$

If in an iteration ρi(h) > ϵ4 then a+h is sufficiently better than a, a is replaced by a+h, and λ is reduced by a factor. Otherwise λ is increased by a factor, and the algorithm proceeds to the next iteration.

{For more information, please refer to the [3].}

I began writing a URDF file to define the robot's structure in a way that is understandable for libraries and computer programs:

```
<?xml version="1.0" ?>
<robot name="robot_arm">
    <link name="link_0_OriginLink">
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <geometry>
            <cylinder length="0.02" radius="0.03" />
        </geometry>
    </visual>
    <inertial>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <mass value="1.0" />
        <inertia ixx="0.001" ixy="0.0" ixz="0.0" iyy="0.001" iyz="0.0" izz="0.001" />
    </inertial>
    </link>
    <link name="link_1_base_joint">
    <visual>
        <origin xyz="0 0 0.0465" rpy="0 0 0" />
```

```
        <geometry>
            <cylinder length="0.093" radius="0.02" />
        </geometry>
    </visual>
    <inertial>
        <origin xyz="0 0 0.0465" rpy="0 0 0" />
        <mass value="1.0" />
        <inertia ixx="0.001" ixy="0.0" ixz="0.0" iyy="0.001" iyz="0.0" izz="0.001" />
    </inertial>
    </link>
    <link name="link_2_shoulder_joint">
    <visual>
        <origin xyz="0 0 0.0315" rpy="0 0 0" />
        <geometry>
            <cylinder length="0.063" radius="0.018" />
        </geometry>
    </visual>
    <inertial>
        <origin xyz="0 0 0.0315" rpy="0 0 0" />
        <mass value="1.0" />
        <inertia ixx="0.0005" ixy="0.0" ixz="0.0" iyy="0.0005" iyz="0.0" izz="0.0005"
/>
    </inertial>
    </link>
    <link name="link_3_elbow_joint">
    <visual>
        <origin xyz="0 0 0.0595" rpy="0 0 0" />
        <geometry>
            <cylinder length="0.119" radius="0.016" />
        </geometry>
    </visual>
    <inertial>
        <origin xyz="0 0 0.0595" rpy="0 0 0" />
        <mass value="1.0" />
        <inertia ixx="0.0005" ixy="0.0" ixz="0.0" iyy="0.0005" iyz="0.0" izz="0.0005"
/>
    </inertial>
    </link>
    <link name="link_4_end_effector">
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <geometry>
            <box size="0.02 0.02 0.02" />
        </geometry>
    </visual>
    <inertial>
```

```
        <origin xyz="0 0 0" rpy="0 0 0" />
        <mass value="0.5" />
        <inertia ixx="0.0001" ixy="0.0" ixz="0.0" iyy="0.0001" iyz="0.0" izz="0.0001"
/>
    </inertial>
    </link>
    <joint name="joint_1_base_joint" type="revolute">
        <parent link="link_0_OriginLink" />
        <child link="link_1_base_joint" />
        <origin xyz="0 0 0" rpy="0 0 0" />
        <axis xyz="0 0 1" />
        <limit lower="-0.6981317007977318" upper="1.3089969389957472" effort="10"
velocity="1.0" />
    </joint>
    <joint name="joint_2_shoulder_joint" type="revolute">
        <parent link="link_1_base_joint" />
        <child link="link_2_shoulder_joint" />
        <origin xyz="0 0 0.093" rpy="0 -2.0943951023931953 0" />
        <axis xyz="0 1 0" />
        <limit lower="0" upper="3.141592653589793" effort="10" velocity="1.0" />
    </joint>
    <joint name="joint_3_elbow_joint" type="revolute">
        <parent link="link_2_shoulder_joint" />
        <child link="link_3_elbow_joint" />
        <origin xyz="0 0 0.063" rpy="0 2.617993877991494 0" />
        <axis xyz="0 1 0" />
        <limit lower="-3.141592653589793" upper="0" effort="10" velocity="1.0" />
    </joint>
    <joint name="joint_4_end_effector" type="revolute">
        <parent link="link_3_elbow_joint" />
        <child link="link_4_end_effector" />
        <origin xyz="0 0 0.119" rpy="0 0 0" />
        <axis xyz="1 0 0" />
        <limit lower="-3.141592653589793" upper="3.141592653589793" effort="10"
velocity="1.0" />
    </joint>
</robot>
```

After making some modifications and fixing small measurement errors, I redefined it as a chain in IKPy. The following page contains the code for the chain written in Python.

```
my_chain = Chain(name='robot_arm', links=[
   OriginLink(),
   URDFLink(
     name="base_joint",
     origin_translation=[0, 0, 0],
     origin_orientation=[0, 0, 0],
     rotation=[0, 0, 1],
     joint_type = "revolute",
     bounds = (rad(-80), rad(100)),
   ),


   URDFLink(
     name="shoulder_joint",
     origin_translation=[0, 0, 0.093],
     origin_orientation=[0, rad(-110), 0],
     rotation=[0, 1, 0],
     joint_type = "revolute",
     bounds = (0,rad(180))
   ),

   URDFLink(
     name="elbow_joint",
     origin_translation=[0, 0, 0.07],
     origin_orientation=[0, rad(180), 0],
     rotation=[0, 1, 0],
     joint_type = "revolute",
     bounds = (rad(-180), 0)
   ),
   URDFLink(
     name="end_effector",
     origin_translation=[0, 0, 0.119],
     origin_orientation=[0, 0, 0],
     rotation=[1, 0, 0],
     joint_type = "revolute",

   )
],active_links_mask=[
   False, True, True,True,False

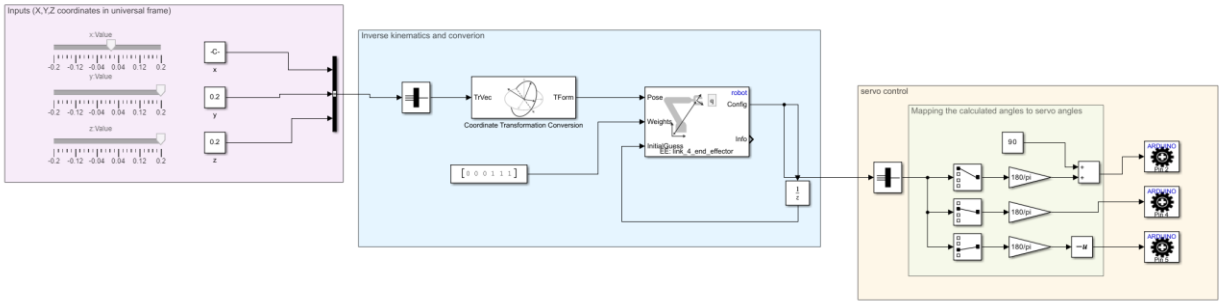


])
```

The servo motors in the robot are equipped with integrated drivers that utilize circular potentiometers for precise position measurement. Each servo is fitted with its own Proportional-Integral (PI) controller, eliminating the need for an external control system at this stage. However, looking ahead to future enhancements, we are considering the addition of a MPU 6050 module to measure disturbances. This would enable us to implement a feedforward controller specifically designed to minimize these disturbances, enhancing the overall performance and stability of the system.

Software Implementation & Computer Vision

Below is a control system designed in Simulink that relies on the robot structure defined by a URDF file. It solves Inverse Kinematics (IK) using the Levenberg–Marquardt algorithm and outputs the calculated angles to the servo motors through an ESP32:



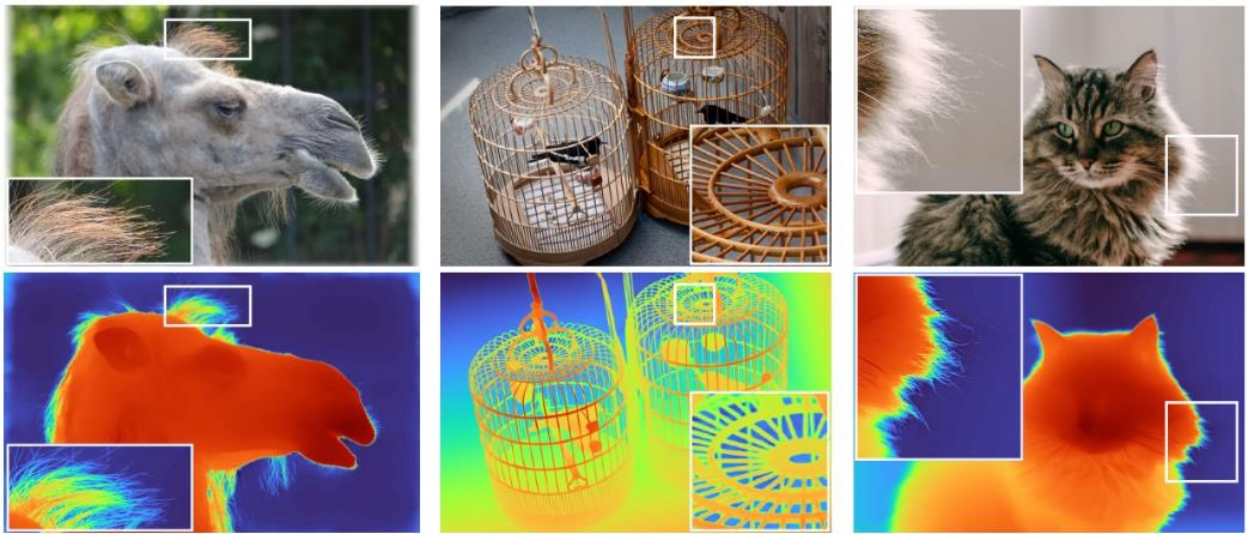
The first block receives inputs when the horizontal bars change. After a coordinate transformation, the inverse kinematics block uses the Levenberg–Marquardt algorithm to solve the inverse kinematics problem and outputs the calculated angles (thetas). These angles are then mapped to the actual servo angles, which are sent to the servos using PWM via the ESP32. The results are relatively accurate, valid, and acceptable.

Use of Camera and Computer Vision:

After solving the inverse kinematics, I transitioned to using a Raspberry Pi as the controller, since the resources of the ESP32 were insufficient to handle everything. I began considering the best way to utilize a camera and computer vision for detecting objects, retrieving their coordinates, and passing that information to the robot control system.

For object detection, my initial thought was to use YOLO v8 and OpenCV. However, I realized that this would overburden the processor. Given that the robot's table is white, I found that simpler methods—such as connected component analysis, contour detection, or finding convex hulls based on color—would be sufficient. I will elaborate on the object detection part later, but first, I want to discuss 3D reconstruction and pose estimation.

To begin, I utilized a depth estimation model to calculate the depth of each pixel in images captured by my webcam. After exploring various depth estimation models available on Hugging Face, I found the DepthPro-hf model developed by Apple. I then proceeded to implement depth estimation in Python, using the Transformers library alongside PyTorch for pre-configurations and to optimize the model's performance on my laptop.



The methodology involved running the model on a laptop with the webcam connected directly to it. After estimating depth, projection, and performing 3D reconstruction, the desired (x, y, z) coordinates were passed to the controller.

The model was generally accurate within a range of millimeters to centimeters. However, I noticed that when I fixed the webcam to focus on the table, changes in environmental lighting caused the model's error to vary visibly and non-linearly. This variability could not be easily corrected. What appeared to be a minor dynamic error became significant after being processed through the rotation and transformation matrices, resulting in a discrepancy of up to 10 cm between the expected position of the robot's end effector and its actual position.

Given the extensive amount of code I have written, I will not include it in this report.

I finally decided to use the flood-fill algorithm in OpenCV. I wrote a program that was functioning well, but I wanted to improve it further. My goal was to detect objects along with their colors in the first step. Since the objects have distinct RGB colors, I implemented color thresholding using ‘cv2.inRange’ in the HSV color space to create masks.

By analyzing the centers of these masks and considering their relative positions in the image, the robot could execute pick-and-place actions from predefined positions to other predefined (x, y, z) coordinates. Here is a snippet of the main program responsible for computer vision:

```
async def Image_processing_task():

    global r_mask_center, g_mask_center, b_mask_center


    vid = cv2.VideoCapture(0)
    vid.set(cv2.CAP_PROP_FRAME_WIDTH, img_w)
```

```
    vid.set(cv2.CAP_PROP_FRAME_HEIGHT, img_h)
    g_width  = vid.get(cv2.CAP_PROP_FRAME_WIDTH)
    g_height = vid.get(cv2.CAP_PROP_FRAME_HEIGHT)
    print(f"Capture resolution: {int(g_width)}×{int(g_height)}")

    for i in range(50):

        ret, frame = vid.read()

    if not ret:
        raise PermissionError("No access to camera!")




    vid.release()




    frame = cv2.undistort(frame, cam_matrix, distortion_matrix, None,
newcameramtx)

    frame = cv2.cvtColor(frame, cv2.COLOR_BGR2LAB)

    frame = frame[roi[1]:roi[3],roi[0]:roi[2],:]
    frame = frame[100:,1000:,:]

    frame[:,:,0] = cv2.equalizeHist(frame[:,:,0])


    rgb_frame = cv2.cvtColor(frame, cv2.COLOR_LAB2RGB)
    hsv_frame = cv2.cvtColor(rgb_frame, cv2.COLOR_RGB2HSV_FULL)

    # -------------
    lower_b = np.array([125, 100, 20])
    upper_b = np.array([184, 255, 255])
    b_mask = cv2.inRange(hsv_frame,lower_b, upper_b)
    b_mask_center = mask_center(b_mask)
    # -------------
    lower_g = np.array([44, 100, 20])
```

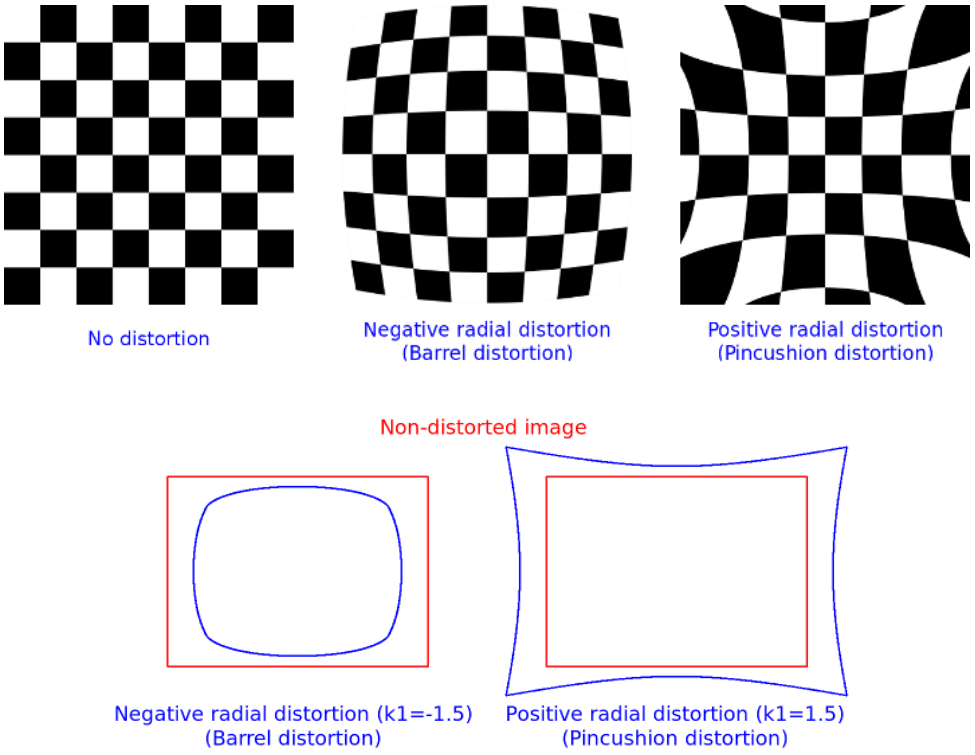
```
    upper_g = np.array([126, 255, 255])
    g_mask = cv2.inRange(hsv_frame,lower_g, upper_g)
    g_mask_center = mask_center(g_mask)
    # -------------
    lower_r1 = np.array([0, 100, 50])
    upper_r1 = np.array([16, 255, 255])
    lower_r2 = np.array([225, 100, 50])
    upper_r2 = np.array([255, 255, 255])


    mask_r1 = cv2.inRange(hsv_frame, lower_r1, upper_r1)
    mask_r2 = cv2.inRange(hsv_frame, lower_r2, upper_r2)
    r_mask = cv2.bitwise_or(mask_r1, mask_r2)
    r_mask_center = mask_center(r_mask)
    # -------------
    b_masked_frame = cv2.bitwise_and(rgb_frame, rgb_frame, mask=b_mask)
    g_masked_frame = cv2.bitwise_and(rgb_frame, rgb_frame, mask=g_mask)
    r_masked_frame = cv2.bitwise_and(rgb_frame, rgb_frame, mask=r_mask)
```

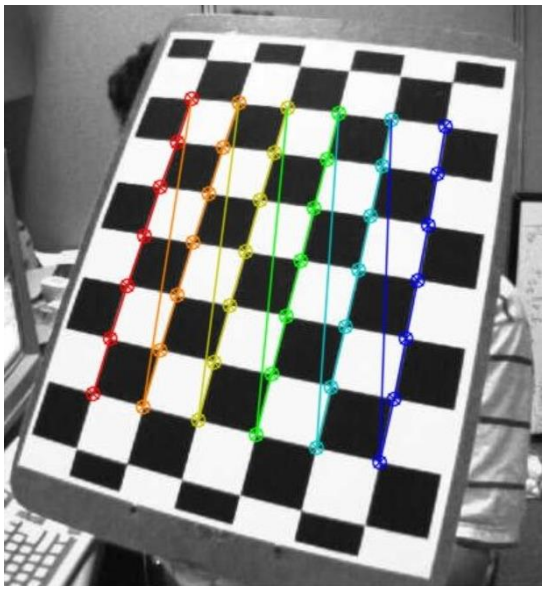
Before processing through the computer vision program, the image is undistorted. This process relates to 3D reconstruction and pose estimation, which I will explain next.

Camera calibration, 3D reconstruction, and pose estimation are key processes that typically occur before object detection. However, for clarity, I have placed this information after the sections on object detection and color thresholding.

Camera calibration involves determining important parameters, such as the camera matrix and distortion vector, which are essential for the subsequent steps in the process. Below are the common types of camera distortion, which primarily include two categories: barrel distortion and pincushion distortion[5].



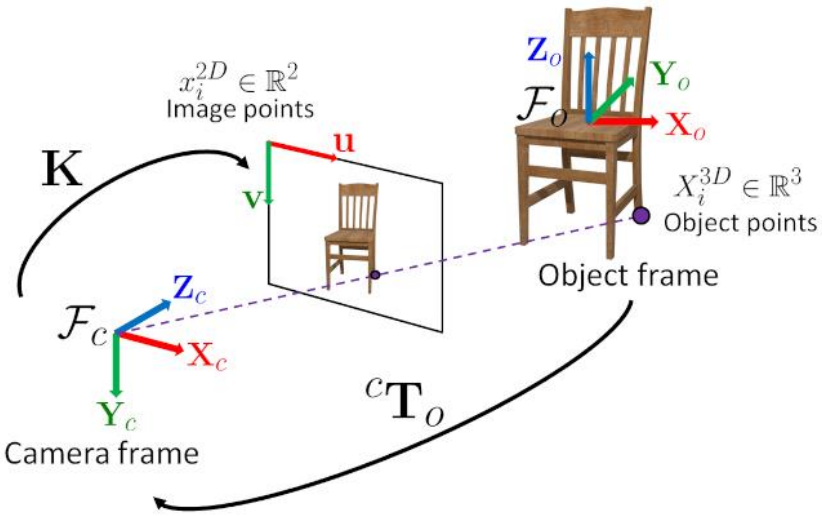
To locate the camera's intrinsic parameters, a chessboard pattern test is conducted. In this test, a printed chessboard pattern is held in front of the camera while capturing multiple images (at least 15) with the pattern in various positions and orientations relative to the camera. Using OpenCV, we can then determine the camera matrix and distortion vector, allowing us to proceed to the next step[6].

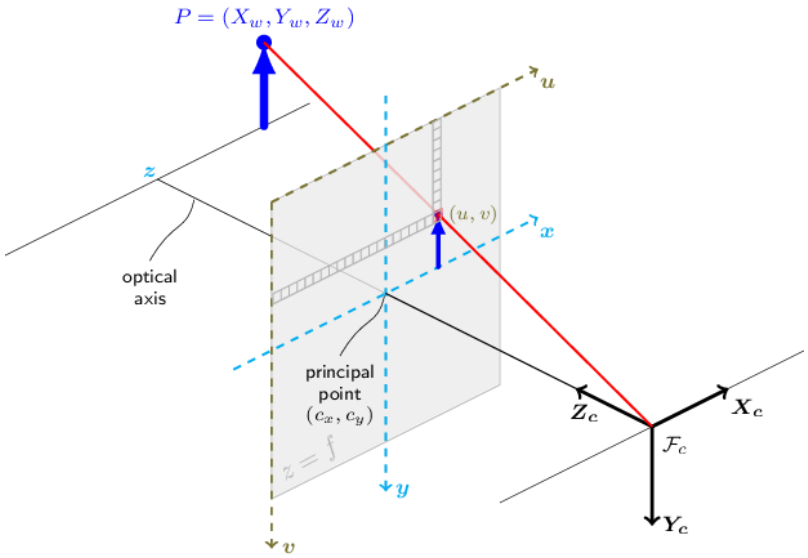


3D reconstruction techniques were used to solve pose estimation, which was relevant when I was employing a depth estimation model. Ultimately, due to simplifications in the problem-solving process, pose estimation was not utilized.

To briefly explain the method: after selecting a pixel in the image and estimating its depth using the model, along with the camera matrix, OpenCV algorithms perform spatial transformations and reproject the data. This process ultimately determines the x, y, and z coordinates of the selected pixel in the camera frame. These coordinates can then be transformed to the robot's universal frame by using a chessboard pattern mounted on the robot's table and applying transformation matrices between the chessboard, the camera, and the robot’s base.

The following section will provide a visual explanation of pose estimation and 3D reconstruction[5].





Conclusion

The robot has successfully demonstrated its ability to accurately pick and place three colored objects with a precision of 1-5mm.

This project encompasses various aspects of robotics, allowing me to explore a range of techniques, from solving inverse kinematics to camera calibration and 3D reconstruction. Each technique has unveiled new scientific principles, deepening my understanding and appreciation of the field. I am excited to continue expanding this project in the near future, enhancing its capabilities and features, and developing it for more advanced and complex tasks.

Bibliography

[1] J. J. Craig, *Introduction to robotics : mechanics and control*. Upper Saddle River, Nj: Pearson, 2018.

[2] Phylliade, “GitHub - Phylliade/ikpy: IKPy, an Universal Inverse Kinematics library,” *GitHub*, Aug. 12, 2024. https://github.com/Phylliade/ikpy/tree/master (accessed Sep. 08, 2025).

[3] H. Gavin, “The Levenberg-Marquardt algorithm for nonlinear least squares curve-fitting problems,” 2024. Available: https://people.duke.edu/~hpgavin/lm.pdf

[4] “apple/DepthPro · Hugging Face,” *Huggingface.co*, Aug. 25, 2025. https://huggingface.co/apple/DepthPro (accessed Sep. 08, 2025).

[5] “OpenCV: Camera Calibration and 3D Reconstruction,” *Opencv.org*, 2025. https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html (accessed Sep. 08, 2025).

[6] “OpenCV: Camera Calibration,” *Opencv.org*, 2025. https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html (accessed Sep. 08, 2025).