Computer Vision Sorting Robotic Arm: Mechanical CAMden

Sam Wisnoski, Satchel Schaivo, Filip Kypriotis

Fundamentals of Robotics with Prof. Kenechukwu Mbanisi

Github: https://github.com/swisnoski/hiwonder-armpi-pro/tree/v2025

AI Disclosure: We used artificial intelligence (specifically ChatGPT) to add comments to sections of our code for readability and for help with LaTeX formatting.

0. Introduction

0.1 Project Overview

For our final project, we aim to mimic the behavior of our close friend, Camden James Droz. Not only is he an inspiration to us all, but more importantly, he is excellent at sorting objects by color. Furthermore, we chose the name "Mechanical CAMden" for it's layered meaning: it is both a tribute to Camden, and a combination of the words "Mechanical" and "Camera". In our report, we refer to our project as "Mechanical CAMden" or just "CAMden" for short.

For our final project, we used computer vision and inverse kinematics to sort cubes of various colors into different bins. While the bins are placed at known locations in space (such that they are hard coded into the robot), we use computer vision and a board of ArUco markers to calculate the proper trajectory of our end effector to lift the cube and place it into the appropriate container.

This requires two distinct processes which we outline below:

1) Computer Vision Analysis: We set the home position of out 5-DOF robotic arm to a position where the camera can clearly see the full ArUco board and all objects placed upon it. Our computer vision algorithm can identify objects and generate their coordinates by using repeated transformations from pixel frame → camera frame → board frame → world frame → robot base frame. Additionally, our CV algorithm will note which color the object is so we can sort it appropriately.

2) Inverse Kinematics and Trajectory Generation: Once we have the coordinates of the object in the robot base frame, we can use inverse kinematics to generate a series of way-points between the current position of the end effector and the position of the object. The arm begins with the claw open, and once it reaches the object, it's claw will close. Since we already know the color from step, it will then generate another path of way-points to above the proper container, drop the object, and return the home position.

These two steps will repeat until every object on the board is sorted. Once it finishes sorting, it will wait until more objects are added to the board.

In our report below, we divide these two steps into four sections:
1. Computer Vision Analysis
2. Kinematics and Robotic Actuation
3. System Integration and Results
4. Conclusion and Reflections

Each of these sections goes into further detail about the math behind these processes, as well as their computational implementation.

0.2 Why is this important?

Our Mechanical CAMden project has broad potential across various real-world settings; CAMden can be used to sort objects by color, shape, labeling, or other visual aspects. In recycling facilities, CAMden could streamline the sorting of plastics, glass, or metals. In agriculture, CAMden could sort produce by ripeness, aiding in quality control and reducing waste. In shipping and logistics, it could sort packages by size or destination, given the ability to read barcodes or labels. With minor modifications, CAMden can be adapted to any facility

where visual inspection and precise sorting are required—especially in systems demanding consistency and speed.

This project is a prime example of how robotics and computer vision have the potential to reshape modern industries. By combining the perception of CV with the motion planning of inverse kinematics, CAMden demonstrates the power of autonomous systems to carry out the jobs that are too repetitive, boring, or unsafe for human workers. However, we recognize the concern that advancements in artificial intelligence and robotics may reduce demand for certain types of labor. While this is true, we believe that as a class project, CAMden is harmless. It's a tool for learning, exploration, and it's just to prove how far a bit of code, a robotic arm, and some clever engineering can make CAMden go.

1. Computer Vision Analysis

In this section, we will discuss the process of extracting object coordinates from an image. We will begin by covering the overall process pipeline, then go into each process individually.

1.0 Computer Vision Pipeline

To understand our full Computer Vision Pipeline, we first need to understand the different coordinate frames we will be translating between:

Our Computer Generation algorithm takes an image from the camera as it's input. We will call this the **Image Frame** or **Pixel Frame**, and it uses the **Image coordinate system** to specify the placement of pixels on an image. This is a two dimensional plane with it's x or u axis specifying horizontal distance from the top left of the image, and it's y or v axis specifying vertical distance from the top left of the image.

Next, we translate the **Image Frame** to the **Camera Frame**, which uses the **Camera Coordinate System**. The origin of the Camera Coordinate System is attached to the camera in the physical world, with the y-axis points downward while it's x-axis points to the right. The z-axis points directly out from the lens, towards the "center" of the Image Frame. It intersects with the image frame at the **principle point**, which has values $(c_x, c_y) = (u_0, v_0)$ in the Image Frame.

Next, we have the **World Coordinate System**. The World Coordinate System includes the Camera, the AcUro Board, and the Robot. The idea is that we know the relationship between the Camera and the Robot Base/Robot End Effector from our IK model, and we know the relation between the Camera, AcUro Board, and objects using our CV algorithm, such that combining this information, we can figure out the position of our desired object in the reference frame of our Robot. Simply, the "World coordinate system relates a camera to other objects in a scene".

So, what does the actual project pipeline look like? Throughout the rest of section one, we will walk you through the following steps:

1. Capture the image and undistort it using known calibration parameters.
2. Identify the ArUco board and calculate the boarder of the board.
3. Identify if there is a red or green object on the board.
4. Convert the image coordinates of an object to the robot base frame.

These four processes are run in a single function, which we can then call from outside the script to return the position of our object in the robot frame:

```
def cv_main(ret, frame0):
    if ret:
        frame1 = undistort(frame0) # Undistort
        frame2 = april_tag_board_corner(frame1) # Draw ArUco board
        frame3 = convert_for_imshow(frame2)
        frame4, ee_position = draw_red_boxes(frame3)

        if ee_position is not None:
            return ee_position
    return None
```

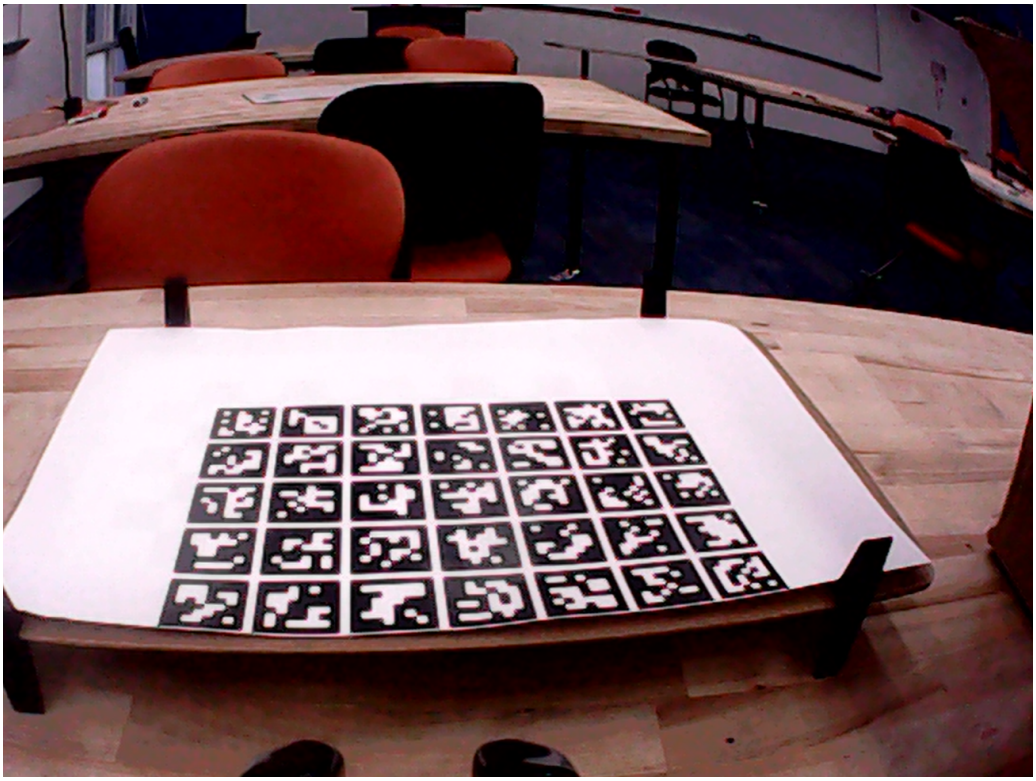
We can also display frame4 to test our detection program and help with debugging if needed.

1.1 Image Capturing, Calibration, and Undistortion

The first step in processing an image is to actually capture the image. Computationally, this is as simple as using the OpenCV library cv2 to connect to the camera and read the current frame.

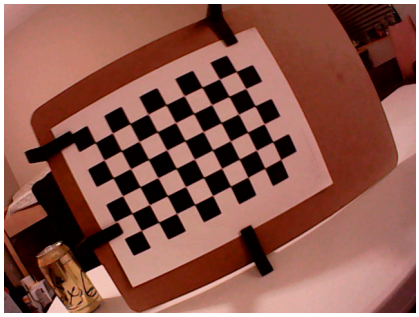
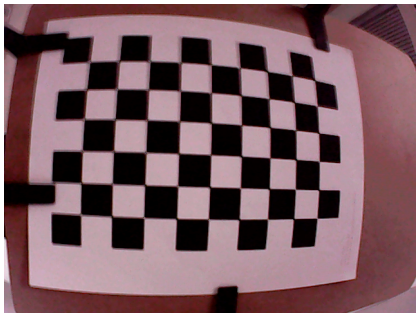
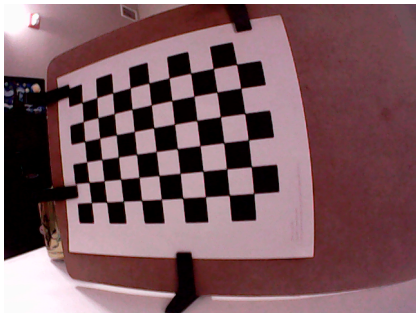
```
video_id = 0
cap = cv.VideoCapture(video_id)
ret, frame0 = cap.read()
```

However, the camera we use to capture the image is a fisheye camera, meaning it automatically distorts the frame in order to capture a wider view. Rather than getting a clear, flat picture of our ArUco board, we would see something like this:



As you get further from the center of the image, the sides have increased distortion, as seen in the table grain, clipboard, and whiteboard. Although possible, this distorted image is significantly harder to process than an undistorted one.

To undistort the image, we first calibrate the camera to quantify its distortion. This is done by taking pictures of a known object, (in our case, a checkerboard), and seeing how it distorts in space across several different images. We can use the CentralCamera.images2C function from Python's machinevisiontoolbox library to complete this goal. First we take a series of images showing the checkerboard from different angles and distances. We used a total of seventeen images, and three are displayed below for reference.

Once we had captured our images, we fed them into CentralCamera.images2C function, along with the shape and size of our checkerboard.

```
### CALIBRATION

images = ImageCollection("./calibration_imgs/*.png")
K, distortion, _ = CentralCamera.images2C(images, gridshape=(9, 6), squaresize=30e-3)

u0 = K[0, 2]
v0 = K[1, 2]
fpixel_width = K[0, 0]
fpixel_height = K[1, 1]
k1, k2, p1, p2, k3 = distortion
```

From the CentralCamera.images2C, we get two variables, K and distortion. K is a 3x3 matrix that contains all of the camera's internal parameters and thus is known as the camera intrinsic matrix. This includes the focal length f_x,f_y and principal point coordinate c_x,c_y. The variable distortion holds our five distortion coefficients defining the distortion of the lens, where k_i is a radial distortion coefficient and p_i is a tangential distortion coefficient.

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \qquad \text{distortion} = k_1, k_2, p_1, p_2, k_3$$

We then extract each of these parameters as shown in the code above. The extracted values are below.

$$K = \begin{bmatrix} 532.6 & 0 & 248.6 \\ 0 & 532.5 & 269.5 \\ 0 & 0 & 1 \end{bmatrix}, \quad (f_x, f_y) = (532.6, 532.5), \quad (cx, cy) = (248.6, 269.5)$$

$$\text{distortion} = [k_1, k_2, p_1, p_2, k_3] = [-0.521, 0.351, -5.3 \times 10^{-4}, 4.87 \times 10^{-5}, -0.144]$$

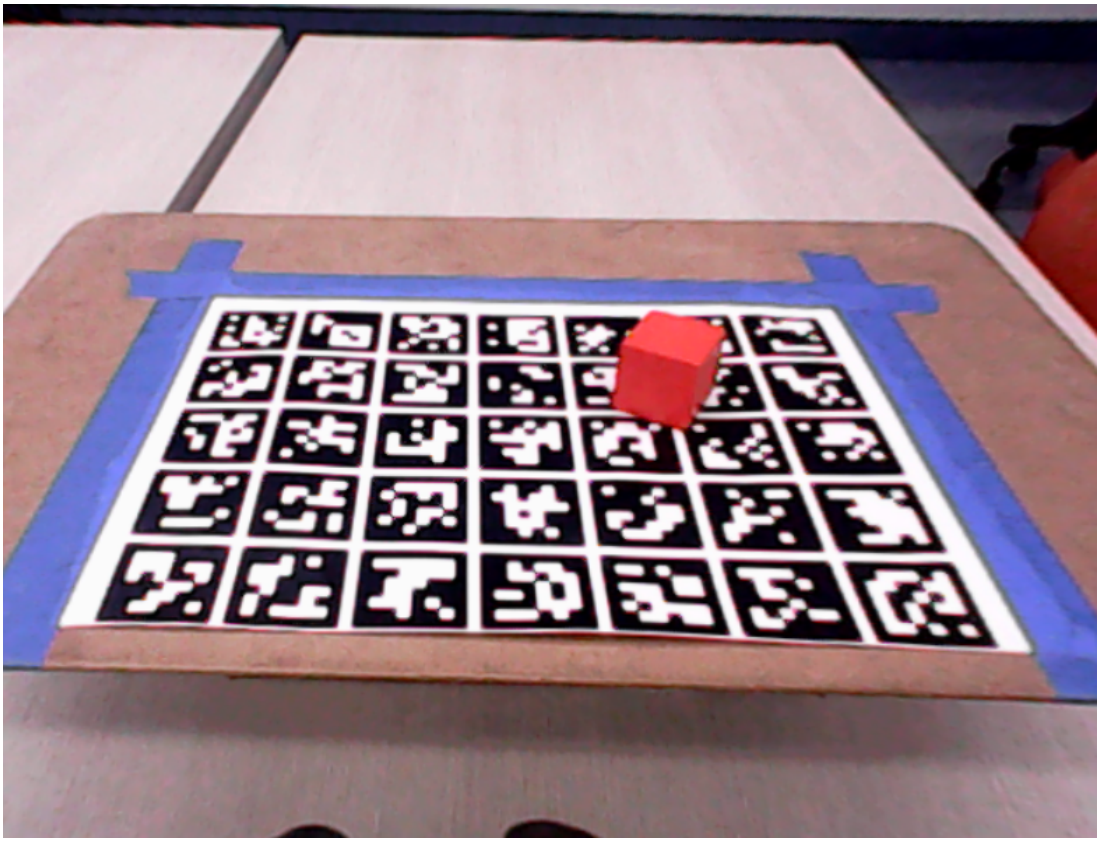
$$k_1 = -0.521, \quad k_2 = 0.351, \quad k_3 = -0.144, \qquad p_1 = -5.3 \times 10^{-4}, \quad p_2 = 4.87 \times 10^{-5},$$

When we first captured an image, it appeared to have a heavy positive radial distortion, which we can confirm with our radial coefficients (k_i) being roughly four magnitudes bigger than our tangential coefficients (p_i). To undo the radial distortion, we use our parameters in the following function:

```
### UNDISTORT FUNCTION
def undistort(frame):
    frame = Image(frame, colororder='BGR')  # Make sure it's interpreted correctly
    U, V = frame.meshgrid()
    x = (U - u0) / fpixel_width
    y = (V - v0) / fpixel_height
    r = np.sqrt(x**2 + y**2)
    delta_x = x * (k1*r**2 + k2*r**4 + k3*r**6) + 2*p1*x*y + p2*(r**2 + 2*x**2)
    delta_y = y * (k1*r**2 + k2*r**4 + k3*r**6) + p1*(r**2 + 2*y**2) + p2*x*y
    xd = x + delta_x
    yd = y + delta_y
    Ud = xd * fpixel_width + u0
    Vd = yd * fpixel_height + v0
    return frame.warp(Ud, Vd)  # Returns an Image object
```

First, we feed in the current frame from our camera, converting it to an image with the color order blue-green-red. Then, we find the pixels of our image (U, V) using the meshgrid function and convert those pixels into normalized x and y coordinates using the principle point coordinate ($u_0 = c_x, v_0 = c_y$) and our pixel dimensions (f_x = pixel width, f_y = pixel height). Next, we calculate the radial distance, r of each pixel from the optical center (the center of the lens) using the Pythagorean theorem. Finally, to undistort, we calculate a δy and δx using our $x/y/r$ distances and our distortion coefficients. We can use the δy and δx to undistort by adding it to our original x and y, and then remap it to image space by multiplying by the pixel height/width and centering the pixels. Lastly, we return the undistorted frame as an object for further processing.

We display an undistorted frame below for reference. The radial distortion has clearly been removed.



1.2 AcUro Board Pose and Border

With our image undistorted, we can now begin detecting objects in our image. The first thing we will reference will be our AcUro Board, which anchors all our object to the camera.

An AcUro board is a type of marker board that consists of square fiducial markers, each with a unique pattern that is easily detected by a camera. This allows us to determine the board itself as well as the position of objects on the board, even if the board is partially obscured.

We begin by using the machinevisiontoolbox to create an ArUcoBoard object. We define the gridshape (how many squares wide and tall), the size of the squares, as well as the spacing between the squares. Although we don't use it immediately, we also calculate the total height, length and corner locations of the board. This is to help up convert from the image frame pixels to world frame distances. Lastly, we create a camera object, once again using the intrinsic matrix (K) to allow for calibration. This code is shown below.

```
gridshape = (5, 7)
square_size = 32e-3
spacing_size = 3.2e-3
boardheight = (gridshape[1]*square_size + (gridshape[1]-1)*spacing_size)
boardwidth = (gridshape[0]*square_size + (gridshape[0]-1)*spacing_size)
board_corners = np.array([
        [0, 0, 0],
        [0, boardheight, 0],
        [boardwidth, boardheight, 0],
        [boardwidth, 0, 0]
    ], dtype=np.float32)

board = ArUcoBoard(gridshape, square_size, spacing_size, dict="6x6_1000", firsttag=0)
C = np.column_stack((K, np.array([0, 0, 1])))
est = CentralCamera.decomposeC(C)
camera = CentralCamera(f=est.f[0], rho=est.rho[0], imagesize=[480, 640], pp=est.pp)
index = 0
```

Next, we can continue using the machinevisiontoolbox to search for an ArUco board. We pass in our undistorted frame and use the estimatePose function to detect an ArUco board in the image. If we find an ArUco board, we estimate the pose of the camera in reference to the ArUco board and we use the draw/draw_rectangle

functions to detect and display the axis and border of the board. Regardless of if we find the board or not, we pass on an image for further processing so we don't error out the code.

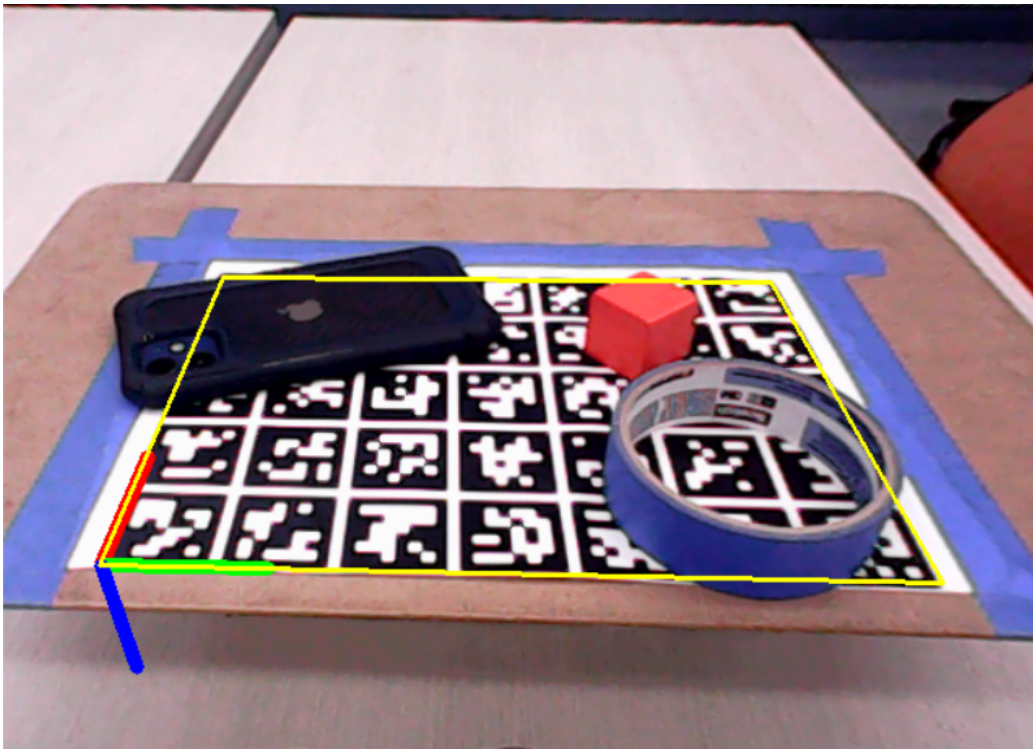
```
def april_tag_board_corner(frame):
    img = Image(frame, colororder='BGR')  # MVTB expects BGR
    try:
        pose_found = board.estimatePose(frame, camera)
        if pose_found:
            pose = pose_found[0]
            board.draw(img, camera, length=0.05, thick=8)
            img = draw_rectangle(img, pose)

    except:
        return img
    return img # Return numpy image (BGR)
```

For the sake of this project, let's pretend we DO see an ArUco board in our frame. In this case, we begin by generating the pose of the board compared to the camera, which is crucial for converting from the board frame to the camera/robot frame. Additionally, we want to calculate the border of the board. This is especially important, since we only want to detect objects that are in the parameters of the board. To draw a border, we use our draw_rectangle function, shown below.

```
zero_dist = np.zeros_like(distortion)
board_contour=None

def draw_rectangle(img, pose):
    global board_contour
    R = pose.R  # 3x3 rotation matrix
    tvec = pose.t.reshape(3,1)
    rvec, _ = cv.Rodrigues(R)
    # Project 3D points to 2D image plane
    imgpts, _ = cv.projectPoints(board_corners, rvec, tvec, K, zero_dist)
    imgpts = imgpts.astype("int32").reshape(-1, 2)
    board_contour = imgpts.reshape(-1, 1, 2)

    # Draw rectangle on the image
    img = cv.polylines(img.image, [imgpts], isClosed=True, color=(0, 255, 255), thickness=2)

    return img
```

The draw_rectangle function takes the current frame and the camera/board pose as arguments. First, we use the pose to calculate the rotation and translation matrices between the board and the camera, flattening the rotation matrix from a 3×3 to a 3×1 matrix. We then project the 3-D board corners into the 2-D plane to display them on the image. Notably, although we use the intrinsic matrix for this calculation, we use zero distortion, since our image and board points are already undistorted. Lastly, we convert our image points into pixel coordinates, save them in the board_contour variable (for later use), and use them to draw yellow lines around the acuro board. We then return the image to the april_tag_board_corner function, where it is passed on for further processing.

We display a frame below with the rectangle and axis drawn the ArUco board for reference. Although a block, my phone, and tape are covering the board, we can still estimate it's pose and border.



1.3 Color Detection and Image Position

After detecting the AcUro Board, we can then begin detecting objects on the board. Since we already know what kind of objects we are looking for, we can set parameters for specific colors to search for:

```
# Red low hue range
lower_red1 = np.array([0, 100, 100])
upper_red1 = np.array([10, 255, 255])
# Red high hue range
lower_red2 = np.array([160, 100, 100])
upper_red2 = np.array([179, 255, 255])

lower_green = np.array([35, 10, 5])
upper_green = np.array([90, 255, 255])

target_rgb_map = {
    "red": (230, 50, 50),
    "green": (50, 140, 50),  # Adjust based on your object
}


color_defs = {
        "red": [
            (lower_red1, upper_red1),
            (lower_red2, upper_red2)
        ],
        "green": [
            (lower_green, upper_green)
        ]
    }
```

We use HSV color vectors (Hue, Saturation, and Value) rather than RGB or BGR because it makes it easier to detect a wide range of colors. Specifically, using HSV allows us to detect a color regardless of how bright or shadowed it is, which is much more difficult than in RGB or BGR without complicated thresholds.

We begin our color detection function by feeding in our preprocessed frame, where we have already undistorted and identified the AcUro Board. We convert our image to be represented in HSV, matching our color vectors, and initialize our return variables: out (the frame we will be returning), robo_position (the position of an object in the robotic frame), and block_color (the color of the object we detect).

```
def draw_boxes(frame):
    global block_color
    hsv = cv.cvtColor(frame, cv.COLOR_BGR2HSV)
    out = frame.copy()
    positions = np.empty(shape=[0, 2])
    robo_position = None
    for color, ranges in color_defs.items():
        mask = np.zeros(hsv.shape[:2], dtype=np.uint8)
        for lower, upper in ranges:
            mask = cv.bitwise_or(mask, cv.inRange(hsv, lower, upper))

        target_bgr = np.array(target_rgb_map[color][::-1], dtype=np.float32)

        mask = cv.morphologyEx(mask, cv.MORPH_OPEN, kernel, iterations=2)
        mask = cv.morphologyEx(mask, cv.MORPH_DILATE, kernel, iterations=1)
        contours, _ = cv.findContours(mask, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)
```

Once we initialize these variables, we begin iterating through our possible detection colors. Although the function is configured to easily detect a wider range of colors, we chose to stick with only two to demonstrate proof of concept and make debugging easier. For each color, we create a blank mask, then redefine it as a binary mask using our predefined color range(s), and then filter it to remove noise and fill in the gaps. Finally, get a list of external contours for our current color.

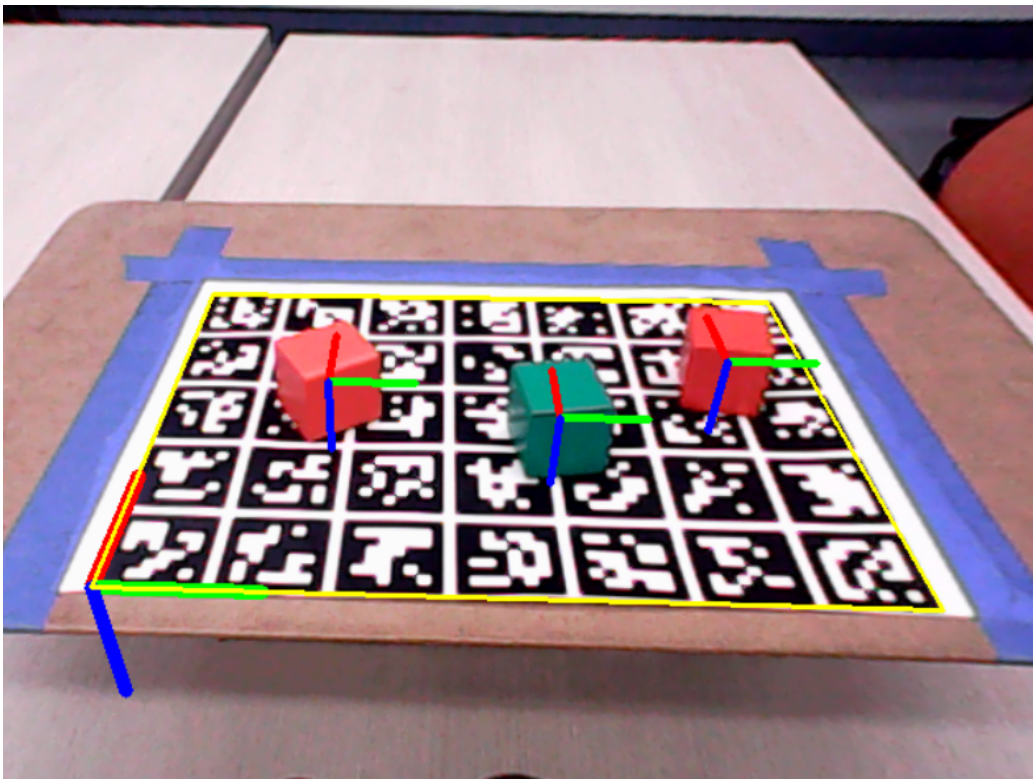
We continue our function by iterating through the detected contours. We begin by filtering out any contours with an area smaller than 500 pixels. If our contour is big enough, we create a mask that is ONLY the contour (compared to the entire image). We then calculate the mean BGR value of the contour and compare it to the target value, which is predefined above and changes with each color. If the contour is close enough to the target value, we then check if it is inside the ArUco Board.

```
        for cnt in contours:
            if cv.contourArea(cnt) < 500:
                continue
            c_mask = np.zeros(mask.shape, np.uint8)
            cv.drawContours(c_mask, [cnt], -1, 255, -1)
            mean_val = cv.mean(frame, mask=c_mask)[:3]
            mean_bgr = np.array(mean_val, dtype=np.float32)
            dist = np.linalg.norm(mean_bgr - target_bgr)
            if dist < 80:
                # Check if all points of the contour are inside the board_contour
                inside = False
                for point in cnt:
                    pt = tuple(int(x) for x in point[0])
                    if board_contour is not None:
                        result = cv.pointPolygonTest(board_contour, pt, True)
                        if result >= 0:
                            inside = True
                            break
                if inside:
                    x, y, w, h = cv.boundingRect(cnt)
                    positions = np.append(positions, [[x + w/2, y + h/2]], axis=0)
                    out, robo_position = calc_object_positions(Image(out, colororder='BGR'), positions)
                    if color == 'red':
                        block_color = "red"
                    if color == 'green':
                        block_color = "green"
                    # cv.rectangle(out, (x, y), (x + w, y + h), (0, 0, 255), 2)

    return out, robo_position
```

We begin with the assumption that object is NOT inside the ArUco board. We then iterate through each point in the contour and compare it's pixel coordinates to our previously calculated ArUco Board border, board_contour. If even a single point of the contour is inside our border, we continue with point calculate with the assumption the object must be at least partially on the board.

If the object is "inside" the border, we get a rough estimate of its image position by taking a rectangle around the image and finding its center. We then pass that to our calc_object_position function, which converts from image coordinates to coordinates in the robotic plane. Additionally, it also draws an axis on each detected object. Lastly, we set the global variable block_color accordingly for use later.

An example board with three objects and their axes is shown below. We should note that we have designed our robot to work with one cube at a time, always targeting the closest cube. Additionally, in the real model, the camera is much closer to the board to allow for the robotic arm to have a greater range of motion.



1.4 Conversion from Image to Robot Frame

As mentioned in section 1.3, one of our last steps is to convert the image coordinates of the objects to robot/end effector coordinates. While we may have made it seem as easy as calling a function, it actually was one of the hardest parts of our project.

It begins with this function, calc_object_positions:

```
R_predefined = np.array([
    [1,  0,  0],   # X right
    [0,  1,  0],   # Y forward
    [0,  0, 1],   # Z down
], dtype=np.float32)

def calc_object_positions(frame, positions):
    img = np.array(frame.bgr, dtype=np.uint8)
    try:
        pose_found = board.estimatePose(frame, camera)
        if pose_found:
            pose = pose_found[0]
            R = pose.R  # 3x3 rotation matrix
            T = pose.t
            tvec = T.reshape(3,1)
            rvec, _ = cv.Rodrigues(R)
            for position in positions:
                u, v = position
                robo_position = pixel_to_board_camera_robot_xy(u, v, pose)
                position_board = pixels_to_board(u, v, pose)
                obj_axis = np.float32([
                    position_board.flatten(),
                    (position_board + R_predefined @ np.array([[0.1], [0], [0]])).flatten(),
                    (position_board + R_predefined @ np.array([[0], [0.1], [0]])).flatten(),
                    (position_board + R_predefined @ np.array([[0], [0], [0.1]])).flatten()
                ])
                imgpts, _ = cv.projectPoints(obj_axis[1:], rvec, tvec, K, zero_dist)
                img = draw_position_axis(img, np.array([[u, v]]), imgpts)
    except:
        return img, None
    return img, robo_position
```

calc_object_positions takes in the current frame and the estimated locations of all of objects on the board. For the sake of explaining our function, let's pretend there is exactly one object on the board. First, we reverify that board is actually still visible in frame and recalculate the pose between the board and the camera, giving us the rotation and translation matrices between the corner of the board and the camera. We actually only use these vectors to draw the axes on the objects later in the function. When we calculate points, we re-pass the entire pose rather than the specific vectors.

When first reading over our function, it may seem like we calculate the position of the objects twice, once in pixel_to_board_camera_robot_xy, and once in position_board. However, the two functions serve different purposes. pixel_to_board_camera_robot_xy is the function that completes our final transformation for the end effector, while position_board simply translates to the board frame to allow us to properly draw our axes on the cubes. Since we already covered very similar code when drawing the rectangle on the ArUco board, we are only going to focus on pixel_to_board_camera_robot_xy in this section.

```
R_cam_to_robot = Rz @ Rx

t_cam_to_robot = np.array([-0.16, 0, 0.23]).reshape(3,1) # 3x1 numpy array

def pixel_to_board_camera_robot_xy(u, v, pose, board_z=-0.02):
    uv1     = np.array([u, v, 1.0]).reshape(3,1)
    ray_cam = Kinv @ uv1

    R, t     = pose.R, pose.t.reshape(3,1)

    ray_b    = R.T @ (ray_cam - t)
    cam_b    = -R.T @ t

    scale = (board_z - cam_b[2,0]) / ray_b[2,0]

    point_board = cam_b + scale * ray_b

    point_cam = R @ point_board + t

    point_robot = R_cam_to_robot @ point_cam
    point_robot = point_robot + t_cam_to_robot
    return point_robot.flatten()
```

Above the function, we define our rotation and translation matrices from the camera to the base of the robot. Since we will always be calculating object position from the home position, we can use static matrices to define this transformation.

The same cannot be said for other transformations. First, we project our pixel coordinate to the camera frame using the inverse intrinsic matrix. We then calculate our transformation vectors from the board to the camera using our generated pose. Next, we compute scale to determine how far along the ray from the camera (the camera's z-axis) we must travel in order to intersect the plane of the board at a known depth (board_z). This scalar tells us the distance from the camera origin, along the direction of the ray, where the point lies on the board surface. Once we have the scale, we can multiply it by the ray vector and add it to the camera's position (in board frame) to obtain the 3D coordinates of the point in the board's reference frame.

Once we have the point in the board frame, we can do two simple transformations, from board to camera, and from camera to robot base, to compute the location of the object in reference to the robot. This is because the object, board, camera, and robot all exist within the world coordinate frame, meaning we can easily translate between them given the transformation matrices.

Lastly, all we need to do is return the point within the robotic frame, and our CV is completed! We can now feed this point into our IK solver in order to compute how to pick it up sort it!

(Notably, we actually pass in both the position of the object ***as well as its color*** *to allow for proper sorting).*

2. Kinematics and Robotic Actuation

The inverse kinematics aspect of this project was both the easiest and most frustrating. Unlike CV, which was completely new to us, we had already worked heavily with both forward and inverse kinematics in Mini-Projects 1 and 2. Because of this, we knew we already had semi-working models and would not have to remake our IK functions. However, we had difficulty with implementing our IK functions reliably and accurately, with

both the functions and the hardware failing at times.

This section of the report will be divided into three sections. Although we already covered kinematics in Report 2, we feel it is important to provide a refresher of the math and functions in order for this to be a complete report. Since we used numerical inverse kinematics, that is what our refresher will cover. Secondly, we will cover the implementation of IK into our pipeline, and thirdly, the difficulties of implementing accurate IK with the robot.

2.1 A Brief Overview of Numerical Inverse Kinematics.

(Note: since we have covered this topic extensively in previous reports and will be using code provided by Kene for our project, we have decided not to use code blocks for this section and just stick to theory.)

In order to calculate numerical inverse kinematics, a few things are first required: the Denavit-Hartenberg Table and a inverse Jacobian Matrix.

To find the DH table for our robot, we can reference Mini-Project 1, where we carefully calculated each of our robot's frames, and then were able to derive our DH table by comparing the position and orientation of each frame.

Frame #	θ_i	d_i	a_i	α_i
Frame 0	θ_1	l_1	0	$\pi/2$
Frame 0.5	$\pi/2$	0	0	0
Frame 1	θ_2	0	l_2	π
Frame 2	θ_3	0	l_3	π
Frame 3	θ_4	0	l_4	0
Frame 3.5	$-\pi/2$	0	0	$-\pi/2$
Frame 4	θ_5	l_5	0	0

After finding our DH table, our next step is to find our inverse Jacobian. The first step to deriving the inverse Jacobian matrix is to derive the generalized Jacobian, characterized as:

$$\begin{bmatrix}\overrightarrow{v}\\ \overrightarrow{\omega}\end{bmatrix} = J(\overrightarrow{\theta})\,\dot{\overrightarrow{\theta}}\text{; where } J = \begin{bmatrix}J_{v_1} & J_{v_2} & \dots & J_{v_n}\\ J_{\omega_1} & J_{\omega_2} & \dots & J_{\omega_n}\end{bmatrix}$$

To find each of our linear Jacobian velocities (J_{v_i}), we can use the formula:

$$J_{vi} = \overrightarrow{z_i} \times \overrightarrow{r_i}$$

To find each of our angular Jacobian velocities (J_{ω_i}), we can use the formula:

$$J_{\omega_i} = \overrightarrow{z_i}$$

To find our linear and angular velocities, we need to derive both the r and z vectors for each frame. Let's start with our z vectors, which are given by the equation:

$$\overrightarrow{z_i} = {}^0R_{i-1} \times \hat{k}\text{, where } \hat{k} = [0,0,1]^T$$

The above z vectors give us the angular velocities, but in order to calculate the full Jacobian, we also need the linear velocities, which require the r vectors. The r vectors are the x, y and z distances between each of our frames and the end effector frame (frame 5). To find each, we simply compute variations our DH table, starting from the first term to the end effector. For each next vector, we remove the top row (or bottom frame) of our DH table. So for example, 0-5 r is the full DH table, and 0.5-5 r is the DH table with the first row removed.

As such, to calculate the value for $\overrightarrow{r_i}$, we need:

$$\overrightarrow{r_i} = \overrightarrow{d_5^0} - \overrightarrow{d_{i-1}^0}$$

Lastly, we can finally find our linear Jacobians using the formula:

$$J_{v_i} = \overrightarrow{z_i} \times \overrightarrow{r_i}$$

Now that we have our Jacobian, we can derive the inverse Jacobian by applying some basic principles of linear algebra. Since our Jacobian has seven columns and six rows, we cannot find the exact inverse, but we can find the best approximation using the formula:

$$J^+ = J^T\left(J\,J^T\right)^{-1}$$

where J^+ is the "pseudoinverse" of J, providing the closest possible approximation of our inverse Jacobian. Now that we have our inverse Jacobian, we can simply plug in our desired position vector to calculate our needed individual joint angles.

Given our inverse Jacobian, we can work on solving the our numerical inverse kinematics. The Newton-Raphson Method is an iterative method, where we take an initial guess, compute error, move in the desired direction, and then repeat until our error is very small. The exact steps of the Newton-Raphson Method are as follows:

1. Take an initial guess. In our case, our initial guess will be the current position of our EE.
2. Calculate the error by subtracting the desired distance and orientation from the current distance and orientation (calculated via Forward Kinematics).
3. If the error is less than our tolerance, ϵ, then we have successfully reached our desired position. We can return our thetas and move to our calculated position.
4. If the error is greater than our tolerance, then we add our error multiplied by the Jacobian to our current thetas. This will move us closer to our desired position. Then, return to step two and repeat.

The Newton-Raphson Method allows us to find the closest possible points (below tolerance), even if they aren't exact. This is perfect for our project, since we need a reliably method to get *close* to our blocks, but it does not need to be perfect.

2.2 Inverse Kinematics Implementation

Not surprisingly, the implementation of inverse kinematics wasn't too bad, since we already had a code base we were familiar with. We implement our inverse kinematics directly in the highwonder pro script, calculating our joint angles and then moving directly to our desired position. This works by initializing a robot object within the hiwonderpro class using the FiveDOFRobot class, by running self.robot = FiveDOFRobot(). Then, when we want to calculate a position, we can simply run the solve_inverse_kinematics function of our FiveDOFRobot. The below script is the function for our first movement, which takes an x and y position, and then moves to a predetermined z above the block. We want to note that get_coordinates **DOES** return an accurate z position as well, but we found that having a set z position worked much more reliably. Sometimes robotics *is* about taking the easy way out, as long as you have a good reason for it.

```
def go_to_position_high(self, pos):
    EE = utils2.EndEffector(pos[0], pos[1], 0.05, self.robot.ee.rotx, self.robot.ee.roty, self.robot.ee.rotz)
    theta = np.degrees(self.robot.solve_inverse_kinematics(EE, tol=0.01))
    theta = np.append(theta, -120)
    if theta[0] > 0:
        theta[0] = theta[0]*2.5
    # print(theta)
    self.set_joint_values(theta)
```

As seen above, we use our generated object coordinates to define an end-effector object, and then feed the end-effector object into our solve_inverse_kinematics function. Then, we convert to degrees, do some minor scaling, and move to the position by calling self.set_joint_values(theta), a built in function to the hiwonderpro class.

Then, we also have an inverse kinematics function to move lower, and hardcoded functions to open the claw, close the claw, and move to above our sorting bins. Our sort function is a great example of a hardcoded movement function, where we simply feed in the color to determine the movement.

```
def sort(self, color):
    if color == 'red':
        theta = [180, 0, 90, -30, 0, 90]

    if color == 'green':
        theta = [-180, 0, 90, -30, 0, 90]
    # print(theta)
    self.set_joint_values(theta)
    return theta
```

And believe it or not, that's our entire inverse kinematics implementation. We do have a few similar functions which can be found on our github (linked at the top of this paper), but as they mirror the functions above almost identically, we chose not to include them. However, as easy as the above implementation seems, we had significant difficulty due to the unreliable nature of both the inverse kinematics and the joint angles of the robot.

2.3 Struggles

As expected with any robotics project, there were a lot of minor hiccups where things went astray. There was lots of debugging the Computer Vision algorithm, messing with color detection and reference frames and random cv2 functions. There were difficulties in coding collaboratively, with merge errors, writing the same code twice, or just having difficulty finding times to meet. However, what was unexpected was the sheer number of hardware/technical issues we ran into when implementing inverse kinematics into the hardware. In this section, I would like to focus on two issues specifically.

First, it took us far to long to debug why our connection to the robot was unreliable and our code would randomly break. Even after two team members attending separate office hours, it took us calling in a CA outside of class/office hours to sit down with us for half an hour and go through each step of the process... only for us to realize we simply had been working on the wrong branch of the hiwonder-pro github. Even though the branch has previously been working for us, something about this project broke it completely, leading to a few unproductive team meetings and many wasted hours debugging. Now, admittedly, this is partially our fault — we should have reached out for help earlier, and we likely missed this information at some point or another, but we still found it frustrating nonetheless. Although this feedback has been loud and clear over the course of the semester, we wanted to once again reiterate it here; more documentation about the libraries/functions we are using, as well as consistency over the course of the semester, would be much appreciated. This is especially true for those who have not taken Software Design or are less familiar with code.

Secondly, once we were able to connect to the hiwonder armpi pro robot and test our inverse kinematics functions, we found that it was very unreliable in it's ability to move to a given point. Although both our object coordinate generation function and our inverse kinematic functions were working well, (or at least as well as possible given the uncertainty of numerical IK), the physical robot would not move to the positions correctly. A 90 degree angle in the IK or visualizer would be closer to 85 or 80 degrees. While this did not heavily affect the x or y positions, as they would generally move smaller angles and the large claw gives room for error, our z measurement was significantly offset. However, when attempting to correct for this by moving our z position lower, our numerical IK often errored out. In the end, we elected to hard code our z position to achieve more accurate results at the cost of versatility. Additionally, we needed to make minor edits to our generated y joints in order to ensure the robot would move enough. Again, we do not believe this was a fault of our CV or IK algorithms, as we measured both to ensure accuracy, but rather a fault of the physical robot.

In the end, although we struggled with this project, even these frustrations were conducive to our learning and allowed us to grow more in the field of robotics.

3. System Integration and Results

In this section, we will cover our System Integration Pipeline, and go over the final results of our project.

3.1 System Integration

The goal of the system integration was to have the CV function input real-life coordinates of an object, which would be fed into our IK solver, which would then tell the hiwonder robot to move according to those joint calculations. Broadly speaking, the biggest area of concern was validating the coordinates from CV without testing using IK, since we wanted to validate each function independently to assure functionality.

To validate the CV function, we would measure the distance from the camera to the box with a ruler and compare our generated coordinates to the measured coordinates. Although it took some debugging, we were eventually consistently getting measurements with $\approx 90\%$ measured accuracy. This was satisfactory, as the end effector claw is bigger than our desired object and has enough tolerance to compensate for slight differences.

To validate IK, we used the visualizer to choose valid positions, and then used inverse and forward kinematics to check if we could convert a chosen end effector position to joint angles, and then back to the same EE position. Then, we generated object coordinates based on our objects, and tested how close the generated end effector position would be to the desired EE position. In the second case, we had a few scenarios where it would error out, but overall, it worked well. Additionally, since we had already validated our previous IK code, and were using the solution IK code for this project, we had confidence in the fact that it would work.

Lastly, we wanted to validate that feeding joint positions to the hiwonder robot would in fact move it to the correct position. To do this, we used the test.py script found in the scripts folder to feed different join configurations to the hiwonder robot. As detailed above in the "Struggles" section, the robot was very finicky and often did not move to the correct location. However, after studying it's movements and noticing it had the

same, repeated inconsistencies, we were able to roughly calibrate it to move to the correct position when over the ArUco board.

With each aspect of our project validated, we were ready to build the full pipeline. To do so, we created a new python script, sorter.py, and imported each of the needed parts: the get_coordinates function as our computer vision algorithm, the FiveDOFRobot for inverse kinematics calculations, and the HiwonderRobot class for robot control.

```
from hiwonder import HiwonderRobot
import numpy as np
from time import sleep

from camera_cv import get_coordinates
from arm_models import FiveDOFRobot
```

Next, we initialized our Hiwonder robot and defined our main function with a series of functions and movements.

```
robot = HiwonderRobot()

def main():
    while True:
        print("Moving to Home")
        robot.move_to_home_position()
        sleep(1)

        print("Getting Coords")
        x, y, z, block_color = get_coordinates()
        sleep(1)

        print("Moving Above")
        robot.go_to_position_high([x,y])
        sleep(1)

        print("Moving On")
        theta = robot.go_to_position_low([x,y])
        sleep(1)

        print("Closing")
        robot.close(theta)
        sleep(2)

        print("Sorting")
        theta = robot.sort(block_color)
        sleep(1)

        print("Opening")
        robot.open(theta)
        sleep(2)

if __name__ == "__main__":
    main()
```

First, we moved to the home position, guaranteeing we would have a good view of the board, and putting our camera in a know position in relation to the robot. Next, we repeatedly scanned for object coordinates and color with our get_coordinates function. Once we had obtained coordinates, we would feed them into a series of movements: moving above the object, moving to the object, closing the claw, moving it to the correct sorting location, and opening the claw. Since this functionality is repeated in an unending while loop, it will then move back to the home position and attempt to scan for more coordinates.

While most of the functions here are explained or shown above, the get_coordinates function is actually a new function build for the pipeline in order to repeatedly search for objects and properly return both position and color. Additionally, it makes sure it doesn't move to the same location twice if there is not a new object that had been placed, a common bug of our CV algorithm.

```python
video_id = 0
cap = cv.VideoCapture(video_id)
ee_position = None

last_coords = None

def get_coordinates():
    sleep(2)
    global last_coords
    global ee_position
    index = 0
    while True:
        ret, frame = cap.read()
        ee_position = None
        ee_position = cv_main(ret, frame)
        if ee_position is not None:
            print(ee_position)
            index += 1
            x, y, z = ee_position
            x = -(x-0.04)
            y = -y
            sleep(1)
            x_rounded = round(x, 2)
            y_rounded = round(y, 2)
            # Check if the rounded coordinates match last_coords
            if last_coords == [x_rounded, y_rounded]:
                print('match!')
                index = 0

            if index > 5:
                ee_position = None
                last_coords = [x_rounded, y_rounded]
                sleep(2)
                return x, y, z, block_color
```

The cv_main function is described above, and implements most of our CV algorithm by calling a series of image processing functions.

With the execution of these new functions, our full pipeline was completed and ready to test. Although we had a decent amount of bug catching to do, we eventually were able to accomplish very satisfactory results, as detailed in the section below.

3.2 Results

Overall, our project was a success. We set out to use computer vision and inverse kinematics to sort objects by color, and at the end of the day, that's what we achieved. Additionally, our robot and script functions well, with around 80% accuracy of picking up an object and 100% accuracy in identifying color. With our initial goal of 75% accuracy, we can concretely say this project was a success. Additionally, when the robot does not correctly pick up the object, it does still get the x and y coordinates incorrect, but the inverse kinematics will occasionally error out and mess up the z coordinate.

To display our robot in action, we created a short video demonstrating the full ability of the robot, which can be found here: https://www.youtube.com/watch?v=L9ZMtG02WtA. Please watch the video on high volume, as our audio is very quiet.

For a more comprehensive detail of our results and methodology, with a much deeper look into our code, please reference our github: https://github.com/swisnoski/hiwonder-armpi-pro/tree/v2025.

4. Conclusion and Reflection

In our last section, we want to reflect upon the totality of this project: the overall outcomes, what did/did not go well, and how we would improve our approach going forward.

Project Outcomes: Beyond the technical results of this project our whole team were able to gain great experience with our learning goals:

Comprehensive Computer Vision: We wanted to gain a deeper understanding of Computer Vision fundamentals through this project, and we certainly did. We learned about camera calibration matrices, color detection, image/camera coordinate frames, and ArUco boards.

Robust Trajectory Generation: Although a secondary goal, we wanted to learn more about trajectory generation throughout this project. As we developed the IK more it became clear that simply creating a moveto function would not suffice as we had to ensure the end effector would not move the block through pathing to it. To achieve this, we used quintic trajectory generation to have a stable path and created our own movement pipeline

Generalized Robotics: We also wanted to learn more about how to work with robotics and integrate mechanical and software systems. While this project was definitely more on the software side, it was still a great learning experience to create the pipeline between sensors and the physical world.

Limitations: Implementing inverse kinematics proved challenging—in order to consistently make the inverse kinematics work, we needed to confine the join limits to only be able to operate within the scope of ArUco board (or within a few inches outside of it). While this isn't necessarily a limitation for our project, it does mean that our project is scoped to this size Aruco board, and further development of inverse kinematics would be needed for a different joint-design space. Likewise, although we tested and validated the returned coordinates from the Computer Vision function, we found much less error from our IK function when elevating the Aruco board to be at the same height as the base of the robotic arm. Finally the overall pipeline is computationally expensive and will take multiple seconds to run from start to finish, which will need to be improved before we can accurately and quickly move to objects.

Future Improvements: As we touched on above, developing a more robust IK solver is the biggest improvement we would like to make. Without modifications to Kene's original code, we struggled to compute accurate z coordinates, and would move not be able to pick up the object (even with a hard coded z position) around 20% of the time. Creating a more complex solver, especially one that can build off our trajectory generation, would be helpful to improve accuracy. Although it was mitigated in final testing, we still struggled to accurately compute height of our objects without scaffholding, and having a verified solver would help us improve that. Furthermore, a lot of our issues with the project stemmed with the physical movements of the robotic arm—having a more robust robotic arm would be helpful in this department.

Overall, we found success in this project, but not without it's hiccups.

Note from the authors: Dear Teaching Team, Thank you for reading our long (maybe too long) LaTeX reports. Mechanical Camden was a great project to end a great class. Thank you for all the hard work and effort this semester!