Intro to Robotics - Lab

Lab - 03

Student 01: Syed Muhammad Ali, sn07590@st.habib.edu.pk
Student 02: Zaid Bin Khalid, zk08128@st.habib.edu.pk

Problem 1: Task 2.12: Camera Intrinsic Parameters

In computer vision and robotics, intrinsic camera parameters define the internal characteristics of a camera. They are essential for converting 2D image coordinates into 3D real-world coordinates. The key parameters include:

1. **Focal Length** (f_x, f_y)
   - Defines the scale between the real-world scene and the image plane.
   - Measured in pixels and given in both x and y directions.
   - Determines how much the camera magnifies the image.
   - Related to the field of view by:
   $$\text{Horizontal FOV} = 2 \times \tan^{-1}\left(\frac{\text{Width}}{2 \times f_x}\right)$$
   - A higher focal length means a narrower FOV.
2. **Principal Point** (c_x, c_y)
   - The point where the optical axis intersects the image plane.
   - Usually close to the image center.
   - Represents the optical center of the camera.
   - If off-centered, it may indicate an image shift.
3. **Skew Coefficient** (s)
   - Measures the angle between the x and y axes of the pixel coordinate system.
   - For most modern cameras, this value is typically 0, indicating no skew.
4. **Distortion Coefficients** $(k_1, k_2, k_3, p_1, p_2)$
   - Represent radial and tangential lens distortions.
   - Required for correcting fisheye effects or barrel distortions.
   - **Radial Distortion** (k_1, k_2, k_3): Causes barrel (bulging) or pincushion (squeezing) distortion.
   - **Tangential Distortion** (p_1, p_2): Caused by lens misalignment.
   - If values are close to zero, the lens has minimal distortion.
5. **Resolution (Width, Height)**
   - Defines the number of pixels in the x and y dimensions.
   - Specifies the width and height in pixels (e.g., 1920 × 1080).
   - Determines image sensor size.
   - Affects field of view and processing speed.

Problem 2: Task 2.13: AprilTag Detection and Positioning

(a) **Possible Causes for Y-Position Offset in AprilTag Reading**

Our AprilTag detection yielded an x-position within reasonable error margins, but the y-position was off by 4 cm. Several factors could contribute to this discrepancy:

(a) **Camera Calibration Errors** If the camera intrinsic parameters (focal length, principal point, distortion coefficients) were not accurately determined or used, the transformation from image pixels to real-world coordinates may have introduced errors. **Impact:** A miscalibrated principal point shifts detected object positions, affecting the y-coordinate.

(b) **Camera Pose or Mounting Angle Issues** If the camera was not perfectly level or was slightly tilted, the AprilTag appears shifted in the image. Even a small tilt can project the tag further along the y-axis due to perspective effects. **Impact:** Misalignment of even a few degrees can cause systematic y-offset errors.

(c) **Perspective Projection & AprilTag Plane Misalignment** AprilTag detection assumes the tag is lying flat relative to the camera. If the tag was not perfectly aligned with the camera's optical axis, it may introduce errors due to perspective distortion. **Impact:** The detected y-position may be compressed or stretched due to skewed angles.

(d) **AprilTag Detection Noise & Pixel Rounding Errors** AprilTags are detected using feature-based recognition. If the camera resolution or AprilTag detection algorithm has noise, pixel rounding errors can accumulate. **Impact:** Small pixel misalignments can exaggerate the y-coordinate shift after coordinate transformation.

(e) **Camera Field Distortions (Lens Aberration)** If the tag was near the edges of the field of view (FoV), lens distortions can cause misinterpretations of position. Barrel distortion can compress parts of the image, causing nonlinear scaling in y. **Impact:** The distortion could systematically shift y-coordinates by a few centimeters.

(f) **External Interference (Lighting & Reflections)** If the lighting was non-uniform, it might have caused shadows or reflections on the AprilTag. If the AprilTag's black-and-white pattern was overexposed or too dim, detection errors could arise. **Impact:** False feature extraction biases the y-coordinate estimation.

(g) **Inconsistent AprilTag Size or Printing Errors** If the AprilTag was printed with slight scale distortions or was wrinkled, it could mislead the detection algorithm. **Impact:** The algorithm may interpret non-uniform scaling, leading to a y-offset.

Conclusion A combination of these factors likely caused the offset in the y-reading. To address this, several strategies can be used:

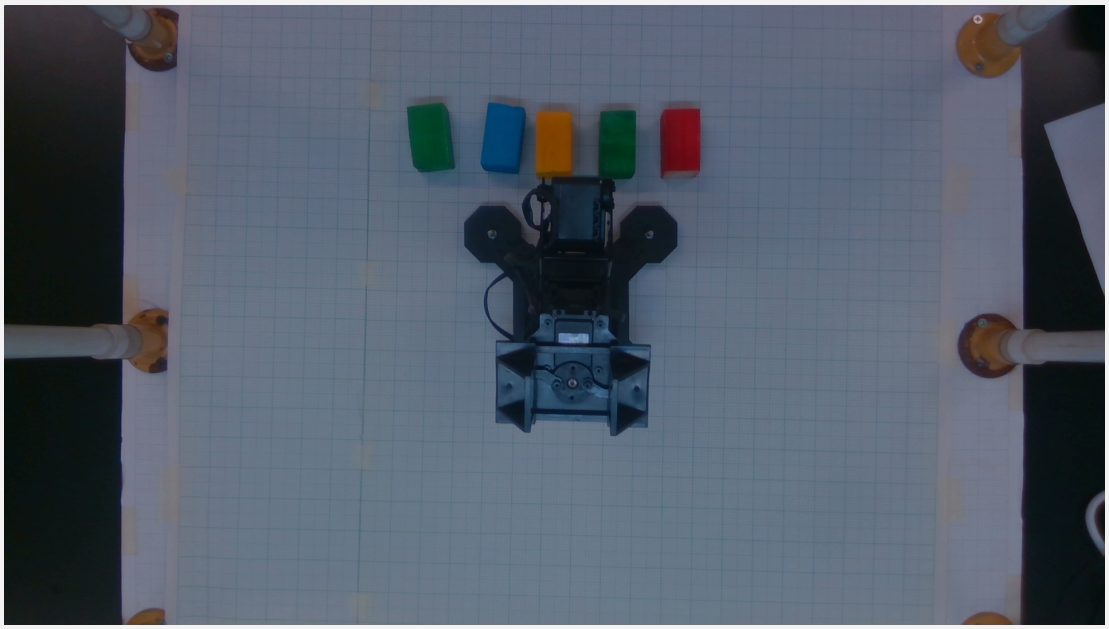
- **Re-check calibration:** Ensure intrinsic values (f_x, f_y, c_x, c_y) are correct.
- **Verify camera tilt:** Adjust extrinsic parameters to align axes.
- **Improve tag placement:** Ensure it is flat & centered in the FoV.

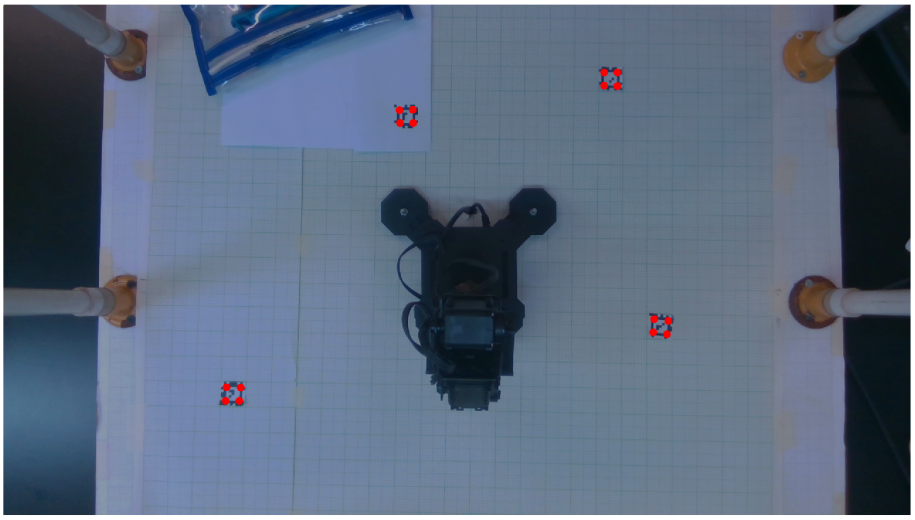
- **Conduct trials with controlled and even lighting** and compare results.
- **Use multiple readings:** Place AprilTags at multiple points and average the readings.
- **Observe consistency:** Check if errors persist, reduce, or new errors arise.
- **Apply radial distortion correction** using the camera's distortion coefficients (k_1, k_2, k_3).
- **Re-run AprilTag detection** near the image center.
- **Use higher-resolution images** to improve edge detection.

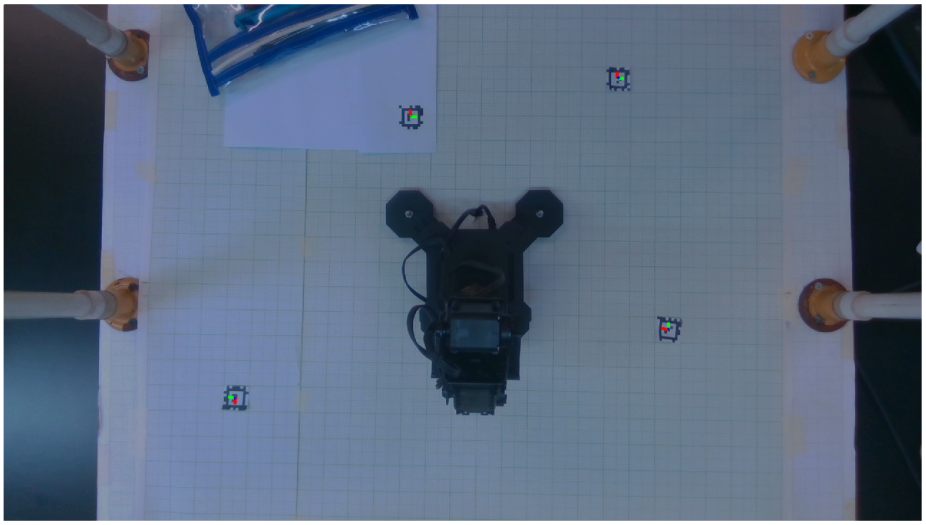
(b)

```
colorintrinsics = determineIntrinsics;
ppx = colorintrinsics.ppx
ppy = colorintrinsics.ppy
fx = colorintrinsics.fx
fy = colorintrinsics.fy


% determineIntrinsics;
% Reading the image
img1 = imread("intro to rob\at3_Color.png");
imshow(img1)

cam_intrinsics = cameraIntrinsics([fx fy],[ppx ppy], [1080 1920]);

[tag, loc,  pose] = readAprilTag(img1, "tagStandard41h12", cam_intrinsics
    , 2.3);

I = img1;
for idx = 1:length(tag)
    % Display the ID and tag family
    disp("Detected Tag ID, Family: " + tag(idx))
        % + ", " ...
        %     + detectedFamily(idx));

    % Insert markers to indicate the locations
    markerRadius = 8;
    numCorners = size(loc,1);
    markerPosition = [loc(:,:,idx),repmat(markerRadius,numCorners,1)];
    I = insertShape(I,"FilledCircle",markerPosition,ShapeColor="red",
        Opacity=1);
end

imshow(I)
```

(c) Raw Image:

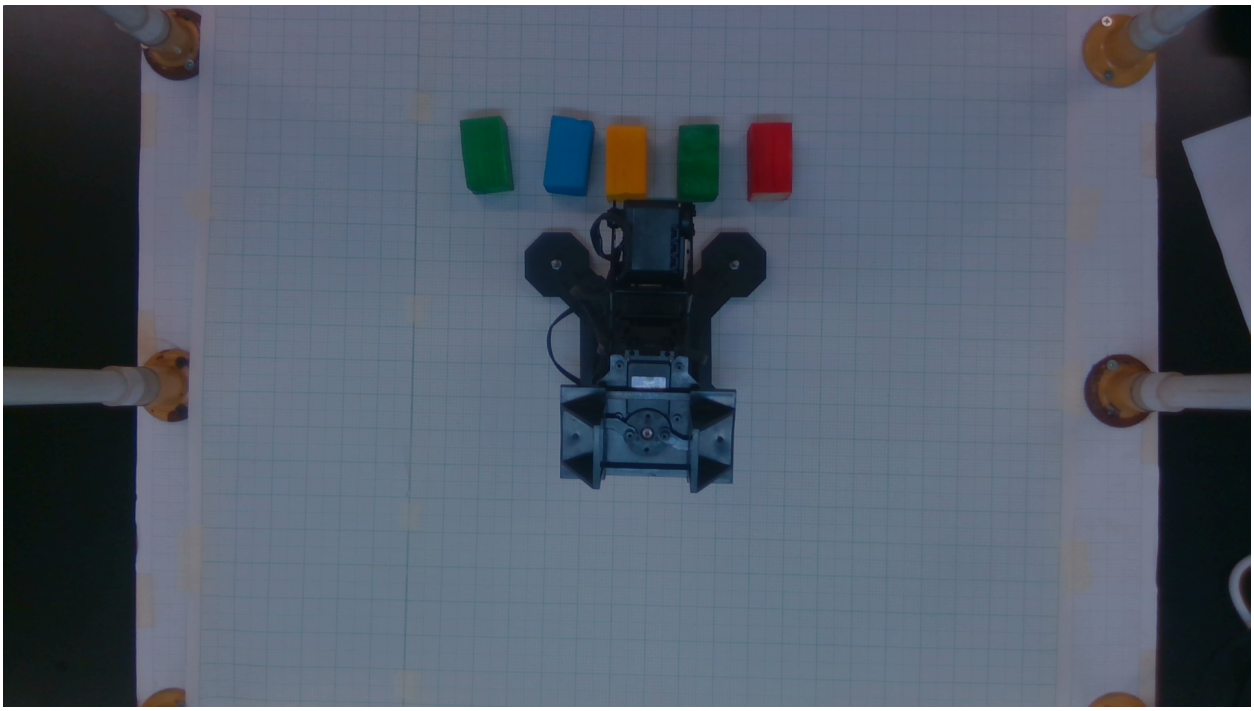






```
colorintrinsics = determineIntrinsics;
ppx = colorintrinsics.ppx
ppy = colorintrinsics.ppy
fx = colorintrinsics.fx
fy = colorintrinsics.fy


% determineIntrinsics;
% Reading the image
img1 = imread("intro to rob\at3_Color.png");
imshow(img1)

cam_intrinsics = cameraIntrinsics([fx fy],[ppx ppy], [1080 1920]);

[tag, loc,  pose] = readAprilTag(img1, "tagStandard41h12", cam_intrinsics, 2.3);

I = img1;
for idx = 1:length(tag)
    % Display the ID and tag family
    disp("Detected Tag ID, Family: " + tag(idx))
        % + ", " ...
        %     + detectedFamily(idx));

    % Insert markers to indicate the locations
    markerRadius = 8;
    numCorners = size(loc,1);
    markerPosition = [loc(:,:,idx),repmat(markerRadius,numCorners,1)];
    I = insertShape(I,"FilledCircle",markerPosition,ShapeColor="red",Opacity=1);
end

imshow(I)
```

Raw Image:



Problem 3: Task 2.14: Camera Datasheet Analysis

The Intel RealSense SR305 depth camera datasheet provides key specifications necessary for robotic perception. Below are the required values along with explanations:

(a) **Resolution of Color Camera and IR Camera**

- **Color Camera Resolution**: 640 × 480 pixels
- **IR Camera Resolution**: 640 × 480 pixels

Explanation:

- The color camera captures standard RGB images.
- The infrared (IR) camera is used to detect depth using the coded light principle.
- Both cameras operate at the same resolution, ensuring accurate pixel-to-pixel depth alignment.

(b) **Frame Rates of Both Cameras**

- **Color Camera Frame Rate**: 30 FPS (frames per second)
- **IR Camera Frame Rate**: 30 FPS

Explanation:

- Frame rate determines how many images are captured per second.
- 30 FPS is typical for real-time processing in robotics applications.
- A higher frame rate ensures smoother motion tracking but requires more processing power.

(c) **Depth Field of View (FoV)**

- **Horizontal FoV**: 69.4°
- **Vertical FoV**: 42.5°
- **Diagonal FoV**: 77°

Explanation:

- The FoV defines the extent of the world the camera captures.
- **Horizontal FoV (69.4°)** → Wider coverage in left-right direction.
- **Vertical FoV (42.5°)** → Determines how much height is visible.
- **Diagonal FoV (77°)** → Overall view coverage.
- These values affect where and how the camera should be mounted.

(d) **Depth Start Point**

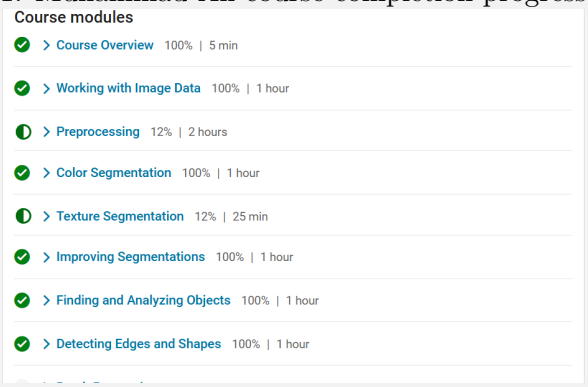
- **Minimum Depth Distance (Near Plane)**: 0.2 meters (20 cm)

Explanation:

- This means the camera cannot detect objects closer than 20 cm.
- If an object is too close, depth readings become inaccurate or undefined.
- The depth accuracy improves as objects move further away.

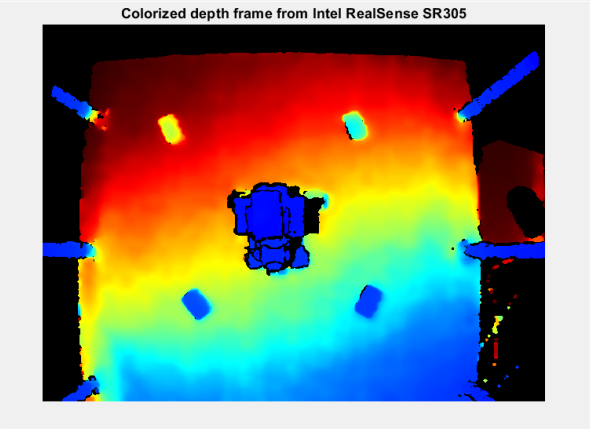
Problem 4: Task 2.15

1. Muhammad Ali course completion progress



Problem 5: Task 2.16

Ran the depth_example code. The figures show the aligned images produced from RGB and depth camera



Problem 6: Task 2.17

1. For frame assignment, we choose to assign the frame to the mid point of the object, with the x-axis aligned with the longer edge of the cube, y-axis along the shorter edge and z-axis pointing upwards from the fact of the object.

   insert figure

2. **Determining Pose from Geometric Features**
   The pose is given as (x, y, z) First, we assume that there is no rotation along x and y axis since they cube is only rotated along the z-axis. This simplifies our problem to finding only 1 angle i.e in z-axis. Moreover, the depth is left to be identified from depth camera. The problem reduces to identifying (x, y, θ) from the geometric features. From geometric features, we identify four corners of the cube. then we proceed as follows

   (a) Average the four points to find the mid point (x, y)
   (b) Covert it to camera frame in pixels and in meters
   (c) Identify any of the longest edge of the cube $(x_1, y_1), (x_2, y_2)$
   (d) Determine which point of the edge has less height. label it pt_1. The other point is labelled pt_2
   (e) Calculate $Y = pt_2(y) - pt_1(y), X = pt_2(x) - pt_1(x)$
   (f) Calculate the angle from the horizontal as $\theta = atan2(Y, X)$
   (g) Calculate the homogenous transform keeping z constant.

3. **Determining Required Geometric Features**
   First we use the image segmentation pipeline to extract relevant pixel blobs. Then we proceed as follows.

   (a) Use region props to extract the convex hull bounding box of each of the box.
   (b) Use Principle Component Analysis to identify the rotation from eigen vectors.
   (c) Rotate the convex hull bbox and idenfity the maximum and minumum points
   (d) Construct a rotated bounding box using the min max points
   (e) The 4 corners of the cuboid are approximately equal to the bounding box corners.

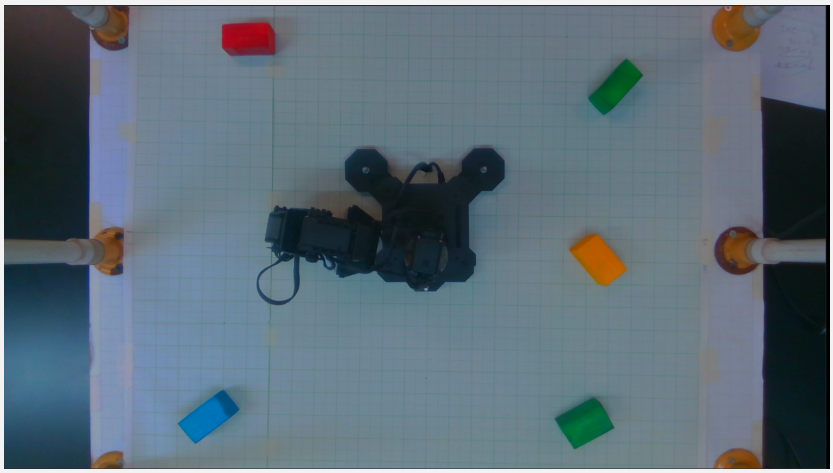
4. Code:

```
        % Find connected components
    stats = regionprops(bw, 'BoundingBox', 'Orientation', 'ConvexHull', '
        PixelList');

    % Display image
    imshow(bw);
    hold on;

    % Store corners
    all_corners = [];
    center_pts = [];
    center_pts_m = [];


for k = 1:length(stats)
    % Get convex hull points (better than bounding box)
    hull_points = stats(k).ConvexHull;

    % Perform PCA for orientation correction
    coeff = pca(hull_points);
    rotated_points = hull_points * coeff; % Align with new basis

    % Find min/max points in rotated space
    min_vals = min(rotated_points);
    max_vals = max(rotated_points);

    % Define rectangle in the transformed space
    rect_pts = [min_vals(1), min_vals(2);
                max_vals(1), min_vals(2);
                max_vals(1), max_vals(2);
                min_vals(1), max_vals(2)];

    % Transform back to original space
    rect_pts = rect_pts / coeff;

     % indetify the higher and lower of teh two points
    if rect_pts(1,2) > rect_pts(2,2)
        edge = [rect_pts(1,:); rect_pts(2,:)];
    else
        edge = [rect_pts(2,:); rect_pts(1,:)];
    end

    % Invert the y cordinates (for beter intuition)
    pt1 = find_pixel_cords([0, 1080], edge(1, :));
    pt2 = find_pixel_cords([0, 1080], edge(2, :));
    pts = [pt1; pt2];

    % dd = norm(pt2 - pt1)
    % angle = acosd(X/dist);

    Y = pts(2,2) - pts(1,2);
    X = pts(2,1) - pts(1,1);
    angle = atan2(Y,X);
    % angle = acosd([X dd]);

    % edge_pts = longer_edge()
```

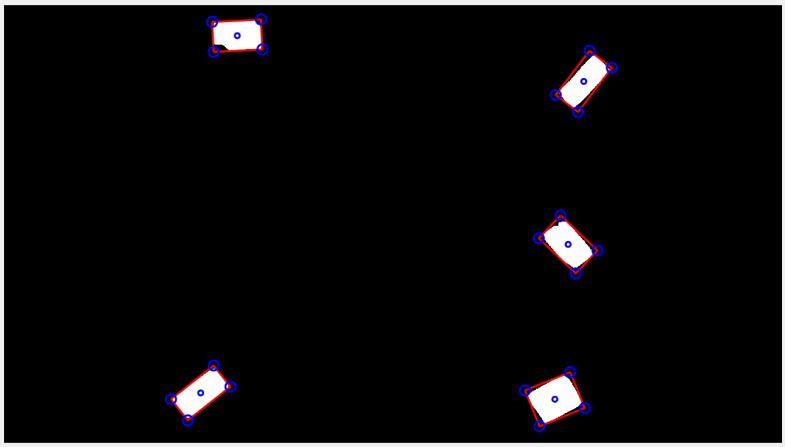
```
    a = mean(rect_pts(:,1));
    b = mean(rect_pts(:,2));

    % Convert center point from pixel to camera cordinates (in pixels)
    pts = find_pixel_cords(cam_center, [a, b]);

    depth = dimg(int32(b),int32(a));

    % comvert the pixel points to meters
    pts_m = pixel_to_meters(pts, [ 0.326 , 0.268 ], [672, 540]);


    center_pts = [center_pts ; pts, angle];
    center_pts_m = [center_pts_m ; pts_m, angle];

    % finding orientation

    % Store corners
    all_corners = [all_corners; rect_pts];

    % Draw the rotated rectangle
    plot([rect_pts(:,1); rect_pts(1,1)], [rect_pts(:,2); rect_pts(1,2)],
        'r-', 'LineWidth', 2);
    plot(rect_pts(:,1), rect_pts(:,2), 'bo', 'MarkerSize', 10, 'LineWidth
        ', 2);
    plot(a, b, 'bo', 'MarkerSize', 5, 'LineWidth', 2);

    end

    hold off;

    function pt = find_pixel_cords(center, image_point)
    x = image_point(1) - center(1);
    y = center(2) - image_point(2);
    pt = [x y];
end

function pose_m = pixel_to_meters(pose_px, half_board_dim, board_edge_px)
    del_x = half_board_dim(1) / board_edge_px(1);
    del_y = half_board_dim(2) / board_edge_px(1);

    pose_m = [pose_px(1)*del_x pose_px(2)*del_y];


end
```

5. The run includes all color objects identified. original image. The images show original image identification of the geometric features, and then the pose estimation. The black line shows x-axis while the white line shows y-axis.

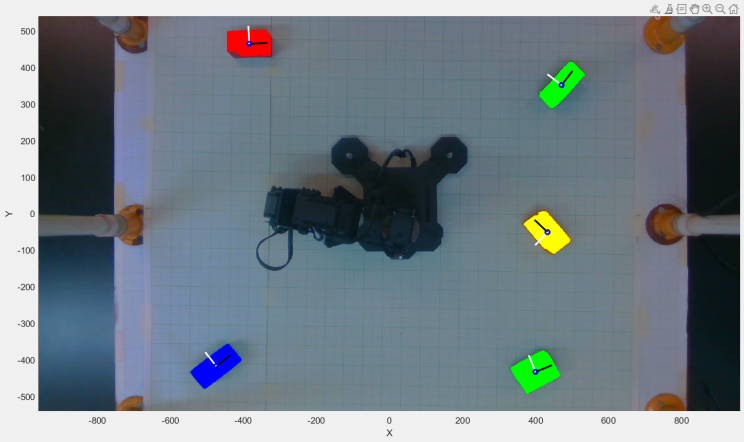
   Original Image



Identification of bounding box



Frame Assignment



6. accuracy test runs

Problem 7: Task 2.18

Below is teh complete pipeline for pose detection of cubes. The pipeline builds on task 2.17 to add depth measurements from the depth image.

```

center = [1920/2, 1080/2, 0]; % camera and depth intrisics se ayen ge ye

[label, rgblabel, overlayingimg, CC, poses, poses_m] = segment_objects("
    intro to rob\test_again_Color.png", "all", "intro to rob\test_bg_Color
    .png", true, false, center);
img = flipud(overlayingimg);
[img_height, img_width, ~] = size(overlayingimg);


% calculating homogenous transform
poses_m

% cTb = homogenous_transform(poses_m)
% cTb2 = homogenous_transform(poses_m(2, :))
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Define image boundaries (centered at (0,0))
x_range = [-img_width/2, img_width/2];
y_range = [-img_height/2, img_height/2];

% Create figure and plot the image centered at (0,0)
figure; hold on;
axis equal; grid on;
xlabel('X'); ylabel('Y');

% Place image with its center at (0,0)
image('CData', img, 'XData', x_range, 'YData', y_range);

% Adjust axis limits
xlim(x_range);
ylim(y_range);

% Define unit vectors for local coordinate frame
axis_length = 50; % Adjust as needed
delta = 1;                          % Distance between consecutive points
l = 50;                                 % Length of square
N = l/delta;                         % Number of points
x_local = zeros(4,N);                       % 1st row has x coordinates and 2
    nd row has y coordinates, 3rd row z coordinates
x_local(1,:) = 0:delta:l-delta;
x_local(2,:) = 0;
x_local(3, :) = 0;                           % Z-coordinate is zero for all
x_local(4,:) = 1; % Converting coordinates to homogenous coordinates
y_local = zeros(4,N);                       % 1st row has x coordinates and 2
    nd row has y coordinates, 3rd row z coordinates
y_local(1,:) = 0;
y_local(2,:) = 0:delta:l-delta;
y_local(3, :) = 0;                           % Z-coordinate is zero for all
y_local(4,:) = 1; % Converting coordinates to homogenous coordinates

% Plot each pose
```

```
    for i = 1:size(poses, 1)
        x = poses(i, 1);
        y = poses(i, 2);
        z = poses(i, 3);
        theta = poses(i, 4);

        % Rotation matrix for 2D rotation around Z-axis
        T = [cos(theta), -sin(theta) 0 x;
             sin(theta),  cos(theta) 0 y;
             0 0 1 z;
             0 0 0 1];

        % Rotate local axes
        x_rot = (T * x_local);
        y_rot = (T * y_local);

        % Translate to pose (x, y)
        x_world = x_rot;
        y_world = y_rot;

        % Plot the coordinate axes
        plot(x_world(1,:), x_world(2,:), 'k', 'LineWidth', 2); % X-axis (Red)
        plot(y_world(1,:), y_world(2,:), 'w', 'LineWidth', 2); % X-axis (Red)
        % plot([x_world(:,2), y_world(:,2)], 'b', 'LineWidth', 2); % Y-axis (
            Green)
        plot(x, y, 'bo', 'MarkerSize', 5, 'LineWidth', 2); % Origin (Blue Dot
            )
    end

    hold off;


function pose_m = pixel_to_meters(pose_px, half_board_dim, board_edge_px)
    del_x = half_board_dim(1) / board_edge_px(1);
    del_y = half_board_dim(2) / board_edge_px(1);

    pose_m = [pose_px(1)*del_x pose_px(2)*del_y];


end


function T = homogenous_transform(pose)
    x = pose(1);
    y = pose(2);
    z = pose(3);
    theta = pose(4);

    T = [cos(theta), -sin(theta) 0 x;
         sin(theta),  cos(theta) 0 y;
         0 0 1 z;
         0 0 0 1];

end
```

```
function [label, rgblabel, overlayimg, CC, center_pts, center_pts_m] =
    segment_objects(img_path, color, bg_path, bgFilter, from_path, cam_center)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% img_path: input file if cam is not live
% from_path: (bool) true if img is to be laoded from live path. False is
% camera is connected
% color: {red,gree,blue,all}: The color that should be detected
% bg_path: (optional): If an image wothout cube is available, it enhance
% segmetation
% bg_Filter (bool): true if bg path is available.
% cam_center: The center of camera in pixels.
    % Reading the image
    if from_path == true
        img1 = imread(img_path);
    else
        [img1, dimg] = get_image();
    end
    % img1 = img;
    % imshow(img1);
    bg = imread(bg_path);

    % Convert images to double precision for accurate subtraction
    bg = double(bg);
    fg = double(img1);

    % Perform subtraction for each RGB channel
    diffR = abs(fg(:,:,1) - bg(:,:,1));
    diffG = abs(fg(:,:,2) - bg(:,:,2));
    diffB = abs(fg(:,:,3) - bg(:,:,3));

    % Normalize differences
    diffR = diffR / max(diffR(:));
    diffG = diffG / max(diffG(:));
    diffB = diffB / max(diffB(:));

    % Create binary masks for each channel using adaptive thresholds
    maskR = diffR > 0.5;  % Adjust threshold as needed
    maskG = diffG > 0.5;
    maskB = diffB > 0.5;

    % Combine masks (logical OR) to detect foreground
    bw_mask = maskR | maskG | maskB;

    % Clean the mask with morphological operations
    bw_mask = imopen(bw_mask, strel('disk', 3)); % Remove noise
    bw_mask = imfill(bw_mask, 'holes'); % Fill small gaps

    % Apply mask to each RGB channel

    % Convert back to uint8
    fg_no_bg = uint8(fg);
    for c = 1:3
        fg_no_bg(:,:,c) = uint8(fg(:,:,c) .* bw_mask);
    end
    img2 = img1;
    if bgFilter == true
```

```
        img1 = fg_no_bg;
    end

    % Converting rgb to LAB
    img1lab = rgb2lab(img1);
    [l a b] = imsplit(img1lab);

    % Converting rgb to HSV
    img1hsv = rgb2hsv(img1);
    [h s v] = imsplit(img1hsv);

    % creating empty masks
    sz = size(img1(:,:,1));
    redmask = zeros(sz);
    yellowmask = zeros(sz);
    greenmask = zeros(sz);
    bluemask = zeros(sz);

    if color == "red" | color == "all"
        redmask = h > 0.9 & s > 0.5 ;
        redmask = imfill(redmask, "holes"); % filling holes
        redmask = bwareaopen(redmask, 200); % size of actual box around 8200
        SE = strel("disk", 3); % creating a disk strel
        redmask = imclose(redmask, SE); % perofrming close operation to
            smoothen edges
        % imshow(redmask);
    end

    if color == "yellow" | color == "all"
        % h extracts color, s extracts regions of yellow with ihgh saturation
            and v
        % ignores the dark yellow areas of the camera stand
        yellowmask = s > 0.5 & h < 0.105 & v > 0.45;
        yellowmask = imfill(yellowmask, "holes"); % as above
        yellowmask = bwareaopen(yellowmask, 200);
        SE = strel("disk", 3);
        yellowmask = imclose(yellowmask, SE);
    end

    if color == "green" | color == "all"
        greenmask = a < -10;
        greenmask = imfill(greenmask, "holes"); % as above
        greenmask = bwareaopen(greenmask, 200);
        SE = strel("disk", 3);
        greenmask = imclose(greenmask, SE);


    end

    if color == "blue" | color == "all"
        bluemask = b < -28 & v > 0.3 & l < 50;
        bluemask = imfill(bluemask, "holes"); % as above
        bluemask = bwareaopen(bluemask, 200);
        SE = strel("disk", 3);
        bluemask = imclose(bluemask, SE);
    end

```

```
    % combining masks

    allcubemask = redmask | greenmask | yellowmask | bluemask;

    % remove the small non-cube noise masked pixels from amsk
    allcubemask = bwpropfilt(allcubemask, "Area", [1000 1080*1920]);
    % find connected components
    CC = bwconncomp(allcubemask);

    % Read binary mask
    % bw = imbinarize(allcubemask);  % Ensure binary

    % Remove noise
    bw = imclose(allcubemask, strel('rectangle', [5,5]));
    bw = imfill(bw, 'holes');

    % Find connected components
    stats = regionprops(bw, 'BoundingBox', 'Orientation', 'ConvexHull', '
        PixelList');

    % Display image
    imshow(bw);
    hold on;

    % Store corners
    all_corners = [];
    center_pts = [];
    center_pts_m = [];


for k = 1:length(stats)
    % Get convex hull points (better than bounding box)
    hull_points = stats(k).ConvexHull;

    % Perform PCA for orientation correction
    coeff = pca(hull_points);
    rotated_points = hull_points * coeff; % Align with new basis

    % Find min/max points in rotated space
    min_vals = min(rotated_points);
    max_vals = max(rotated_points);

    % Define rectangle in the transformed space
    rect_pts = [min_vals(1), min_vals(2);
                max_vals(1), min_vals(2);
                max_vals(1), max_vals(2);
                min_vals(1), max_vals(2)];

    % Transform back to original space
    rect_pts = rect_pts / coeff;

     % indetify the higher and lower of teh two points
    if rect_pts(1,2) > rect_pts(2,2)
        edge = [rect_pts(1,:); rect_pts(2,:)];
    else
        edge = [rect_pts(2,:); rect_pts(1,:)];
    end
```

```
272
273     % Invert the y cordinates (for beter intuition)
274     pt1 = find_pixel_cords([0, 1080], edge(1, :));
275     pt2 = find_pixel_cords([0, 1080], edge(2, :));
276     pts = [pt1; pt2];
277
278     % dd = norm(pt2 - pt1)
279     % angle = acosd(X/dist);
280
281     Y = pts(2,2) - pts(1,2);
282     X = pts(2,1) - pts(1,1);
283     angle = atan2(Y,X);
284     % angle = acosd([X dd]);
285
286     % edge_pts = longer_edge()
287     a = mean(rect_pts(:,1));
288     b = mean(rect_pts(:,2));
289
290     % Convert center point from pixel to camera cordinates (in pixels)
291     pts = find_pixel_cords(cam_center, [a, b]);
292
293     depth = dimg(int32(b),int32(a));
294
295     % comvert the pixel points to meters
296     pts_m = pixel_to_meters(pts, [ 0.326 , 0.268 ], [672, 540]);
297
298     pts = [pts depth];
299     pts_m = [pts_m depth]
300
301
302
303
304     center_pts = [center_pts ; pts, angle];
305     center_pts_m = [center_pts_m ; pts_m, angle];
306
307     % finding orientation
308
309     % Store corners
310     all_corners = [all_corners; rect_pts];
311
312     % Draw the rotated rectangle
313     plot([rect_pts(:,1); rect_pts(1,1)], [rect_pts(:,2); rect_pts(1,2)], 'r-'
        , 'LineWidth', 2);
314     plot(rect_pts(:,1), rect_pts(:,2), 'bo', 'MarkerSize', 10, 'LineWidth',
        2);
315     plot(a, b, 'bo', 'MarkerSize', 5, 'LineWidth', 2);
316
317     end
318
319     hold off;
320
321     % finding labels
322     label = bwlabel(allcubemask);
323
324     % converting labels to rgb
325     rgblabel = label2rgb(label, "jet","k", "shuffle");
326
```

```
    % Viewing the color segmentated image
    k1 = imoverlay(img2, redmask, "red");
    k2 = imoverlay(k1, greenmask, "green");
    k3 = imoverlay(k2, bluemask, "blue");
    overlayimg = imoverlay(k3, yellowmask, "yellow");

    % overlayimg = labeloverlay(img1, label);


end

% modify to make it in meters

function pt = find_pixel_cords(center, image_point)
    x = image_point(1) - center(1);
    y = center(2) - image_point(2);
    pt = [x y];
end
```

References

https://www.leorover.tech/shop/phantomx-pincher-robot-arm-with-adapters

https://github.com/IntelRealSense/librealsense/wiki/Projection-in-RealSense-SDK2.0.

https://www.intelrealsense.com/wp-content/uploads/2019/07/RealSense$_S$R30$x_P$roduct$_D$atasheet$_R$ev$_0$02.*pdf*