Pseudo Code

get_perspective_warping_with_aruco.py

1. Start the webcam and show the live camera feed in a window

2. Wait for the user to click four points on the camera image

3. While the user is clicking, draw small dots on the image to show the chosen points.

4. Once four points have been clicked:

- Convert the clicked pixel points into an array.

- Create another array containing the corresponding real-world table coordinates (X, Y) in millimeters

5. Use OpenCV's getPerspectiveTransform() to compute the 3×3 homography:

H = perspective transform that maps (u, v) pixels to (X, Y) table frame.

6. Apply the homography to the live camera image using warpPerspective() to produce a top-down table view

7. Show the transformed image so the we can verify that the mapping looks correct.

8. When the we press 'q':

- Save the final perspective matrix H into a file named perspective_matrix.npy so other scripts can load it later.

- Close the windows and stop the camera.

aruco_detection_test.py

Described below in image_frame_to_table_frame and marker_centers_in_table_frame

block_detection_aruco.py

constructor (init)- Sets up the node: loads camera calibration, loads the perspective matrix, creates subscribers/publishers, and prepares storage for detected marker data.

crop_frame- Optionally trims away the edges of the image to remove irrelevant background and focus on the workspace.

load_camera_calibration- Reads the YAML calibration file and extracts the camera intrinsic matrix and distortion coefficients for pose estimation.

detect_aruco- Locates ArUco markers in the image, draws their borders, and estimates their pose so axes can be drawn on top of them.

get_aruco_center- Computes the pixel center of each detected marker by averaging its four corners and draws a dot on the image.

image_frame_to_table_frame and marker_centers_in_table_frame:

For each detected marker index i:

1. Read the pixel location (u, v) of the marker center from marker_centers[i].

2. Form a 3-element column vector representing this pixel point in homogeneous coordinates:
   [u]
   [v]
   [1]

3. Multiply the 3x3 perspective matrix H by this 3x1 vector.
   This produces a new vector:
   [x']
   [y']
   [w']
   which is the non-normalized table-frame point.

4. Extract w' from the result.
   This is the denominator that accounts for the perspective distortion.

5. Divide x' and y' by w' to obtain the actual table-frame coordinates:
   X = x' / w'
   Y = y' / w'
   These (X, Y) values represent the marker position in the table coordinate system

6. Store (X, Y) into marker_centers_in_table_frame[i] so the rest of the pipeline knows where the marker is in the real workspace.

7. Convert X and Y into strings so they can be drawn on the image.

8. Draw three lines of text on the image:
   - The ArUco ID number
   - The X coordinate in mm
   - The Y coordinate in mm

   Each marker is printed in its own row so we can visually confirm the coordinates update in real time.

image_callback- Runs every time a new camera frame arrives: detects markers, computes their centers, maps them to table coordinates, stores results, and publishes the annotated image and debug text.

get_latest_detections- Returns the most recently detected IDs and their table-frame coordinates so other scripts (e.g., block-moving logic) can use them.

main- Starts ROS2, creates the ArucoTracker node, loads calibration files, and keeps the node running until shutdown.

kinematic_functions.py

This code involves forward and inverse kinematic algorithms to determine what location the end-effector should end up in and what angles and velocities the joints should achieve to move to that position.

1. Given a set of UR3e joint angles, build the overall 4x4 transformation matrix from the robot base to the gripper using DH parameters.
2. Given a desired gripper position and yaw in the world/table frame, compute an elbow-up set of six joint angles (IK) that should put the gripper at that pose.

calculate_dh_transform(joint_positions)

High level idea:
This function takes in the six joint angles of the UR3e and returns a single 4x4 matrix that tells us where the end effector is in space relative to the base. It uses the standard DH approach: define link parameters, build a 4x4 matrix for each link, and multiply them all together.

1. Input:

- We give the function an array of 6 joint positions:
  joint_positions = [q1, q2, q3, q4, q5, q6]

2. Define the DH parameters:
   - There are 7 transforms in total (including base and tool offsets).
   - For each of the 7 "links," define:
     alpha[i] = twist angle between z-axes
     r[i] = link length along x
     d[i] = link offset along z
   - These are hard-coded numerical values chosen for the UR3e geometry:
     alpha = [0, -pi/2, 0, 0, -pi/2, pi/2, 0]
     r = [0.150 * sqrt(2), 0, 0.244, 0.213, 0, 0, 0.0535]
     d = [0, 0.162, 0.027, 0, 0.104, 0.083, 0.151]
3. Build the effective joint angles for each link:
   - The robot's mechanical zero and the DH zero do not match, so the code uses offsets.
   - It takes the 6 input joint angles and creates a list of 7 theta values:
     theta[0] = *3pi/4 (fixed offset)*
     theta[1] = -3pi/4 + q1 (joint 1 + offset)
     theta[2] = q2 (joint 2)
     theta[3] = q3 (joint 3)
     theta[4] = q4 - pi/2 (joint 4 + offset)
     theta[5] = q5 (joint 5)
     theta[6] = q6 (joint 6)
4. Initialize the overall transform:
   - Start with T as the 4x4 identity matrix.
     This means "no rotation, no translation" at the beginning.
5. Loop through each of the 7 links:
   For i from 0 to 6:
   - Compute:
     ct = cos(theta[i])
     st = sin(theta[i])
     ca = cos(alpha[i])
     sa = sin(alpha[i])
   - Build the 4x4 DH transform A for this link:
     A(0,0) = ct
     A(0,1) = -st * ca
     A(0,2) = st * sa
     A(0,3) = r[i] * ct
     A(1,0) = st
     A(1,1) = ct * ca
     A(1,2) = -ct * sa
     A(1,3) = r[i] * st

     A(2,0) = 0

A(2,1) = sa
A(2,2) = ca
A(2,3) = d[i]

A(3,0) = 0
A(3,1) = 0
A(3,2) = 0
A(3,3) = 1

- Multiply this into the running transform:
  T = T * A

6. The idea is that each A is the transform from one frame to the next; multiplying them accumulates all the rotations and translations down the arm.
   Final result:
   - After the loop, T is the final base-to-gripper (tool) transform.
   - The function returns this 4x4 matrix as "transform".

inverse_kinematics(xWgrip, yWgrip, zWgrip, yaw_WgripDegree)

High level idea:
This function takes a desired gripper location and yaw angle in the world/table frame and computes a specific elbow-up joint configuration [theta1..theta6] that should put the gripper there. It does this by converting world coordinates to the UR3 base frame, finding the wrist center (where the wrist joint is, not the suction tip), solving for the base joint angle, solving for the elbow and shoulder angles using geometry (law of cosines), and finally solving for wrist angles.

1. Inputs:
   - xWgrip, yWgrip, zWgrip: gripper coordinates in world/table frame (meters).
   - yaw_WgripDegree: gripper yaw in degrees (rotation about vertical axis).
2. Define link lengths:
   - The function defines 10 geometric constants L1 through L10 for the UR3e:
     L1 = 0.152
     L2 = 0.120
     L3 = 0.244
     L4 = 0.093
     L5 = 0.213
     L6 = 0.104
     L7 = 0.083
     L8 = 0.092
     L9 = 0.0535
     L10 = 0.059
   - These represent offsets and segment lengths measured along the arm.
3. Convert yaw from degrees to radians:

- yaw = yaw_WgripDegree * (pi / 180)

4. Step 1: Convert gripper world coordinates to base-relative coordinates and set theta5:
   - The world coordinates that come from the camera / table are not the same as the UR3 base frame, so the function applies fixed offsets:
     xWgrip = xWgrip + 0.15
     yWgrip = yWgrip - 0.15
     zWgrip = zWgrip - 0.01
   - These values shift the coordinate system so that (xWgrip, yWgrip, zWgrip) are now relative to the robot base.
   - The wrist bend angle is fixed for this project:
     theta5 = -pi / 2
   - This keeps the wrist in a consistent orientation, making the IK simpler.
5. Step 2: Compute the wrist center (x_cen, y_cen, z_cen):
   - The gripper tip is not at the wrist center; the wrist is some distance "behind" the tip along the yaw direction.
   - Subtract the final small link length L9 along the yaw orientation:
     x_cen = xWgrip - L9 * cos(yaw)
     y_cen = yWgrip - L9 * sin(yaw)
     z_cen = zWgrip
   - (z does not change here because this small offset is in the horizontal plane.)
6. Step 3: Solve for theta1 (base rotation):
   - First compute the polar angle from the base to the wrist center:
     alpha = atan2(y_cen, x_cen)
   - Then compute a small geometric offset for the base joint based on link heights:
     theta_x = arcsin((L2 - L4 + L6) / sqrt(x_cen^2 + y_cen^2))
   - The base joint angle theta1 is:
     theta1 = alpha - theta_x
   - Intuition: alpha points to the wrist center, and theta_x adjusts for how the robot structure is built.
7. Step 4: Solve for theta6 (final wrist rotation):
   - Once we know the base angle and the desired gripper yaw, the final wrist joint just makes up the difference
   - The relation is:
     theta6 = pi/2 - yaw + theta1
   - This orients the gripper to match the desired yaw, given how joints 1 and 5 are set.
8. Step 5: Find the target point for the elbow chain (x_3end, y_3end, z_3end):
   - We remove the effect of the final small links (L7 and L6) from the wrist center:
     x_3end = x_cen - cos(theta1) * L7 + sin(theta1) * (0.027 + L6)
     y_3end = y_cen - sin(theta1) * L7 - cos(theta1) * (0.027 + L6)
     z_3end = z_cen + L10 + L8
   - These formulas shift the wrist center back through the robot's wrist geometry to get the "end" position that the elbow chain (links L3 and L5) must reach.

9. Step 6: Solve for theta2, theta3, and theta4 using triangle geometry:
   a) Compute vertical and horizontal distances from joint 2:
   - The base of the long elbow chain is at height L1.
   - Vertical difference:
     delta_z = z_3end - L1
   - Horizontal radial distance in the x-y plane:
     r_xy = sqrt(x_3end^2 + y_3end^2)

10. b) Compute a preliminary angle theta2a:
    - theta2a = atan2(delta_z, r_xy)
    - This angle is the direction from joint 2 to the elbow chain endpoint, without accounting for link lengths.

11. c) Compute distance c from joint 2 to the chain endpoint:
    - c = sqrt(x_3end^2 + y_3end^2 + delta_z^2)
    - This is the straight-line distance from joint 2 to the point (x_3end, y_3end, z_3end).

12. d) Solve for elbow angle theta3 using the law of cosines:
    - Using L3 and L5 as sides of the triangle, and c as the opposite side:
      cos_term = (L5^2 + L3^2 - c^2) / (2 * L3 * L5)
    - Then:
      theta3 = pi - arccos(cos_term)
    - Using pi minus the arccos gives the elbow-up configuration.

13. e) Solve for an additional shoulder contribution theta2b:
    - Using the sine rule:
      theta2b = arcsin(L5 * sin(pi - theta3) / c)

14. f) Combine to get theta2:
    - theta2 = -theta2a - theta2b
    - The signs are chosen to give an elbow-up solution that matches the robot's convention.

15. g) Solve for wrist pitch angle theta4:
    - We want the total rotation about the middle joints to line up properly, so:
      theta4 = -theta3 - theta2

16. Collect all joint angles:
- theta1, theta2, theta3, theta4 were solved above.
- theta5 was fixed at -pi/2.
- theta6 was computed from yaw and theta1.
- Pack them into an array:
  return_value = [theta1, theta2, theta3, theta4, theta5, theta6]

11. Output:

- The function returns this array of six joint angles as the elbow-up IK solution for the given gripper pose.

main_pipeline.py

spin_all- Loops over all nodes in the global NODES list and calls rclpy.spin_once on each, allowing all callbacks (camera, joint states, etc.) to be processed without a blocking spin.

UR3eController.__init__- Sets up the UR3e controller node: defines a “home” joint configuration, stores current joint angles and gripper state, creates publishers/subscribers for robot commands, positions, and gripper input.

UR3eController.position_callback- Updates the stored joint angles (self.thetas and self.current_position) whenever a new PositionUR3e message arrives; flags that a valid joint state has been received.

UR3eController.input_callback- Reads the gripper’s digital input bits from GripperInput and stores a simple flag self.digital_in_0 indicating whether vacuum is detected.

UR3eController.move_arm

Create a new CommandUR3e message called msg.
Set msg.destination equal to the dest list that was passed into move_arm (These are the six desired joint angles in radians.)

Set msg.v equal to self.vel.
Set msg.a equal to self.accel (These define how fast and how quickly the joints should move.)

Set msg.io_0 equal to self.gripper_toggle_state. (This keeps the gripper in whatever on/off state it is currently in; move_arm does NOT change the gripper state.)

Publish msg on the UR3e command topic using self.pub_command.

UR3eController.gripper_control

Create a new CommandUR3e message called msg.

Set msg.destination equal to self.current_position (This tells the robot to stay in its current joint configuration and not move the arm while we toggle the gripper.)

Set msg.v equal to self.vel.

Set msg.a equal to self.accel. (These are still required even though we are not changing the pose.)

Set msg.io_0 according to the toggle_state argument. (If toggle_state is True, turn the vacuum on; if False, turn it off.)

Update self.gripper_toggle_state to the same toggle_state value. (This keeps the controller's internal record of the gripper state in sync.)

Publish msg on the UR3e command topic using self.pub_command.

UR3eController.at_goal- Compares the current joint angles to a desired destination and returns True if the absolute error in every joint is smaller than a small tolerance.

UR3eController.goal_error- Returns a list of absolute differences between current and desired joint angles, useful for printing out when the arm times out.

BlockMover.__init__- Stores references to the UR3e controller and ArUco tracker, the list of desired destination poses, and sets a safe intermediate Z height used to avoid collisions when moving.

BlockMover.move_block

Print out the initial_position for debugging.

Compute a "pre-initial" joint configuration:
  Use inverse kinematics ik(...) with:
    x = initial_position[0]
    y = initial_position[1]
    z = self.intermediate_height (a safe height above the block)
    yaw = initial_position[3]
  Store this joint solution in pre_initial.

Compute an "initial" joint configuration at the actual block height:
  Use ik with:
    x = initial_position[0]
    y = initial_position[1]
    z = initial_position[2] (grasp height)
    yaw = initial_position[3]
  Store this in intial (initial).

Compute a "pre-final" joint configuration:
  Use ik with:
    x = final_position[0]

y = final_position[1]
z = self.intermediate_height (safe height over the drop-off spot)
yaw = final_position[3]
Store this in pre_final.

Compute a “final” joint configuration at the final drop height:
Use ik with:
x = final_position[0]
y = final_position[1]
z = final_position[2] (place height)
yaw = final_position[3]
Store this in final.

Sequence of actions:

1) Ensure gripper is OFF:
call ur3e_controller.gripper_control(False)

2) Move above the starting block:
call ur3e_controller.move_arm(pre_initial)

3) Move down to the block at grasp height:
call ur3e_controller.move_arm(intial)

4) Turn gripper ON to pick up the block:
call ur3e_controller.gripper_control(True)

5) Lift the block back to the safe height above start:
call ur3e_controller.move_arm(pre_initial)

6) Move in the air to above the final location:
call ur3e_controller.move_arm(pre_final)

7) Move down to the final place height:
call ur3e_controller.move_arm(final)

8) Turn gripper OFF to release the block:
call ur3e_controller.gripper_control(False)

9) Lift back up to the safe height above the final location:
call ur3e_controller.move_arm(pre_final)

BlockMover.process_blocks

Create a new UR3eController() object temporarily and read its home joint angles
into a variable called home_position.

Call UR3eController().move_arm(home_position).
(This creates a brand new controller instance and asks that instance
to move the arm to home.)

For each index and position in marker_positions:

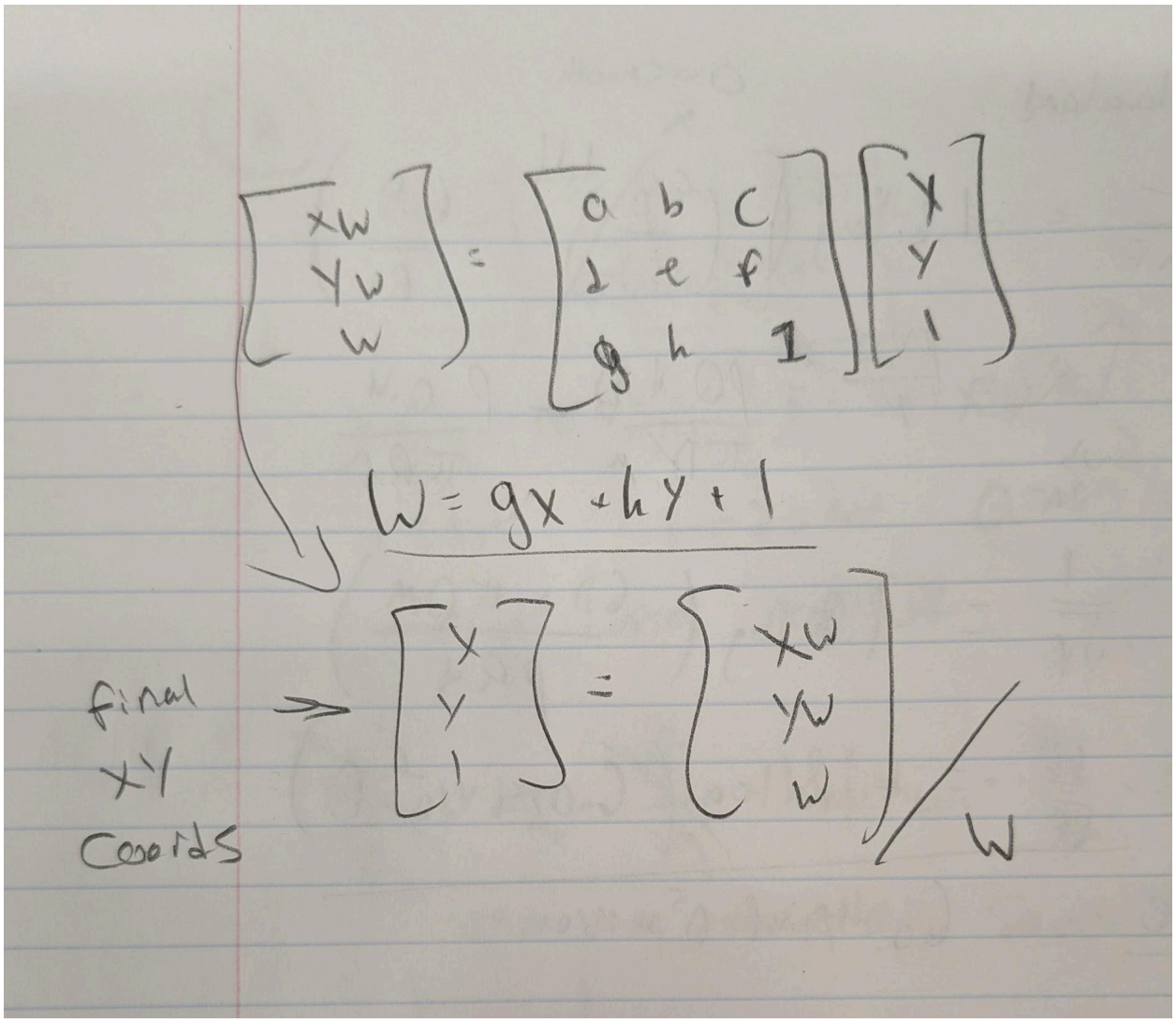
- position is the (X, Y) marker location in the table frame
(units are assumed to be millimeters).

- Build a 4-element list tempsomething:
tempsomething[0] = position[0] / 1000 (convert X from mm to meters)
tempsomething[1] = position[1] / 1000 (convert Y from mm to meters)
tempsomething[2] = 0.048 (fixed Z height for all blocks)
tempsomething[3] = 0 (fixed yaw angle for all blocks)

- Call self.move_block(tempsomething, destination[index])
This moves the block from its detected pose tempsomething
to the corresponding destination pose in the destination list
(the same index as the marker).

main- Initializes ROS2, creates the UR3eController and ArucoTracker nodes, registers them in NODES, waits briefly for the first joint state, defines three destination poses (one per block color/stack height), constructs a BlockMover, and calls process_blocks to start the pick-and-place sequence. Finally it cleans up the node and shuts ROS down.

Math for Camera Frame to Table Frame



Video

https://drive.google.com/file/d/1F9405LX2qNbdLKkDLWBSedykpeXK4-Uv/view?usp=sharing

Pick and Place:
https://drive.google.com/file/d/1xIUf23MvLy5uv_CO7wHNe44Ki5W__aDz/view?usp=sharing