Team 03 EECS 467 Report - ARM LAB

Kun Huang, Saad Mallick, Nico Ramirez

I. Introduction

In this project, we explore the usage of a robotic arm to grab various tagged blocks in a simulated environment. Each block will be grabbed by the robotic arm and returned to a specific bin, depending on the calculated color. We use multiple modern robotics algorithms for the placement of the gripper, localization of the robot in space and movement.

II. Forward Kinematics

Begin by calculating the forward kinematics for the robotic arm (given some joint angles of the motors, calculate the position of the final gripper). To do this, begin by finding some transformation T such that T is the transformation between the frame of the current motor and the previous motor. When T is multiplied by the position of the current motor, it will give its position in the frame of the previous motor, and from here work backwards to find the position of the gripper in the frame of the base. This is considered as the frame of the rex arm.

To determine T, the first three columns of T will be the location of the axis unit vectors of the new frame. The last column is the shift of the axis from the previous set.

Our overall DH matrix is as follows:

	base	shoulder	elbow	wrist
d	0.112	0.055	0.055	0.12
a	0	0	0	0
α	$\pi/2$	0	0	0

Note that the d for base is distance from the ground to shoulder motor.

III. Inverse Kinematics

To solve the inverse kinematics, we use geometrical method to calculate the joint angles for the arm.

The inputs for the inverse kinematics are position x, y, z of the endpoint, and the angle between the end-effector and the horizontal plane ϕ.

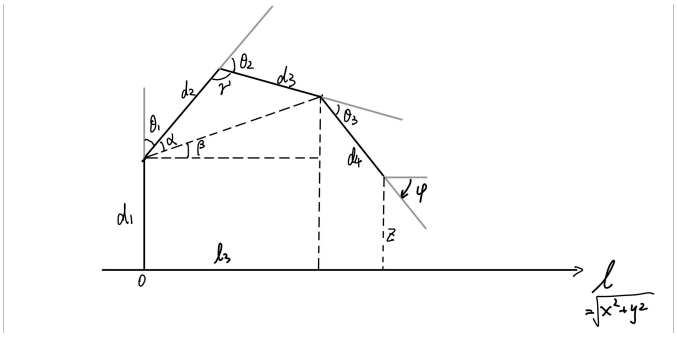

Fig. 1. Notations for all symbols used in the IK solver.

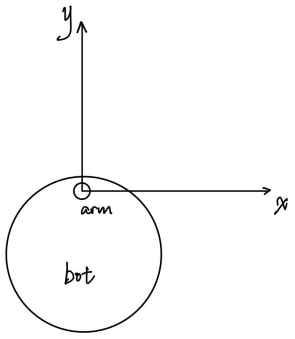

Fig. 2. Coordinate of arm frame.

The angle of the base joint will be

$$\tan^{-1}(\frac{-x}{y})$$

We will then formulate everything in the l, z coordinate.

Since we know ϕ, we can get the position of the wrist motor: $(l_3, z_3) = (\sqrt{x^2+y^2} - d_4\cos\phi, z - d_4\sin\phi)$. Thus, we have

$$\cos\alpha = \frac{d_2^2 + l_3^2 + (z_3 - d_1)^2 - d_3^2}{2d_2\sqrt{l_3^2 + (z_3 - d_1)^2}}$$

$$\beta = \tan^{-1}(\frac{z_3 - d_1}{l_3})$$

through which we can get

$$\theta_1 = \pi/2 - \alpha - \beta$$

Similarly, we have

$$\cos\gamma = \frac{d_2^2 + d_3^2 - l_3^2 - (z_3 - d_1)^2}{2d_2d_3}$$

Thus,

$$\theta_2 = \pi - \gamma$$

We can finally have

$$\theta_3 = \pi/2 - \theta_1 - \theta_2 - \phi$$

IV. Camera Calibration and Object Localization with AprilTags

Methodology

Intrinsic Matrix: To calibrate the intrinsic camera matrix we printed a checkered calibration board and moved it in front of the Raspberry Pi while running the provided calibration program. We had to adjust the size of the checkers in the code to match the size on the printed paper.

Extrinsic Matrix: The extrinsic matrix was determined by solving for a mapping from camera-frame coordinates to rex-arm frame coordinates.

AprilTags were used to localize points in the camera frame. AprilTags were arranged at 4 known locations in the rex arm frame and detected by the AprilTags library. The intrinsic matrix was provided to the library so that it could automatically convert the locations from pixel coordinates to camera frame coordinates. OpenCV's solvePnP function was then used to generate the rotation and translation matrices that describe the transformation from rex arm frame to camera frame. Care was taken to match the corresponding points in the two frames when input into solvePnP.

solvePnP output the transform from the rex arm frame to camera frame but we needed the transform to take coordinates from camera frame to rex arm frame. So we constructed a homogeneous matrix from the output and took the inverse of this homogenous matrix to obtain the extrinsic matrix. This matrix is reported in the results.

Discussion

Intrinsic Matrix: Initially when we calibrated the intrinsic matrix, we only moved the checkered calibration board left, right, up and down. However when we did this the april tag localization was innacurate. We decided to retune the intrinsic matrix by moving the calibration board in more that 2 axis. In addition to left, right, up and down, we moved it towards the camera, away from the camera and skewed it to different angles relative to the camera. This resulted in the camera matrix shown in the results section which allowed us to achieve more accurate AprilTag localization.

Extrinsic Matrix: The arrangement of the points in the rex arm frame was chosen so as to mimic operating conditions. That is we wanted to the calibration points to be approximately where the april tags were expected to show up during operation so that we did not extrapolate too much which might result in innacuracies. The april tags were placed in a stacked square arrangement 6 inches from the origin of the rex arm frame (chosen to be the front stand of the robot). The y-axis of the rex-arm ran along the front-back line of the robot, and the x-axis was parallel to the floor and perpendicular to the y. The arrangement of april tags as seen from the origin of the rex arm frame was the following,

$a \quad b$

$c \quad d$

where a, b, c, d are defined in the results section.

Results

Intrinsic Matrix: The final intrinsic camera matrix:

$$\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 596.1 & 0 & 322.7 \\ 0 & 598.6 & 232.1 \\ 0 & 0 & 1 \end{bmatrix}$$

The corresponding distortion coefficients:

$$\begin{bmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{bmatrix} = \begin{bmatrix} 0.337 & -1.53 & 0.0114 & 0.00398 & 2.63 \end{bmatrix}$$

Extrinsic Matrix: Number of points/april tags used for calibration: 4

(x,y,z) coordinates of calibration points in rex arm frame. All coordinates are in inches:

$$a = \begin{bmatrix} -0.5 & 6 & 1.5 \end{bmatrix}^T$$

$$b = \begin{bmatrix} 0.5 & 6 & 1.5 \end{bmatrix}^T$$

$$c = \begin{bmatrix} -0.5 & 6 & 0.5 \end{bmatrix}^T$$

$$d = \begin{bmatrix} 0.5 & 6 & 0.5 \end{bmatrix}^T$$

Extrinsic matrix:

$$\begin{bmatrix} 1.0 & -0.0044 & -0.0088 & 0.0011 \\ 0.0071 & -0.30 & 0.95 & 0.053 \\ -0.0069 & -0.95 & -0.30 & 0.072 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

V. Color Discrimination

Methodology

Color Discrimination: The color discrimination was accomplished via a 1D closest-neighbor algorithm. The algorithm is provided a set of 'neighbors' or centroids which are hue in the HSV color space. The the distance of a data point with hue value h to each centroid m_i is calculated as $d = \sqrt{(\cos h - \cos m_i)^2 + (\sin h - \sin m_i)^2}$. The color associated with the centroid m_i with the miminum d is output as the classification. In this case the h and m_i values are converted to degrees by multiplying the original values from 0-180 by 2.

Color Localization: To determine where to apply color discrimination in the image, we utilized the pose of the blocks. The AprilTag library output the transform from blocks in the tag frame to the camera frame. We multiplied this matrix by the intrinsic matrix to convert from points in the block frame to pixel locations. The blocks were known to be oriented such that the y-axis in the april tag frame was either facing up or down. Thus we chose the point slightly greater than 0.5 inches above the origin of the april tag frame in the y-axis to sample color from. This point was converted to pixel coordinates and the color discrimination algorithma was applied.

Discussion

In RGB-space, lighting conditions drastically affected the red, green, and blue channels simultaneously. The brightness of illumination was approximately proportional to the values of the 3 channels. However in HSV space, the brightness of the illumination did not directly affect the hue (H) channel. The brightness did affect the value (V) and to a certain extend the saturation (S) channels. Thus for our application we chose the HSV color space since it was much more invariant to illumination changes than RGB. This allowed us to classify colors consistently in different lighting conditions. Since the

hue was mostly invariant to lighting, we were able to use the simple nearest-neighbor algorithm in 1 dimension to classify the color. The 'neighbors' used were the hue value for each color. To make this more robust we averaged the hue over several trials for each color. These values are reported in the results section. Since hue values of 179 and 1 are equally similar as say 69 and 71, we consider the euclidian distance of points as if they lie on a unit circle which is $(\cos h - \cos m_i)^2 + (\sin h - \sin m_i)^2$.

Results

Color space used: HSV

Neighbors/centroids used in the color classification algorithm (hue ranges from 0-180):

red	green	blue	purple	orange	yellow
2.0	69.0	99.0	170.0	10.0	28.5

This table was used to determine, for each colored block, the range of values of hue that correspond to it:

Color	Min H	Max H
Red	176.0 or 0.0	180.0
Orange	6.0	19.25
Yellow	19.25	48.75
Green	48.75	84.0
Blue	84.0	134.5
Purple	134.5	176.0

VI. Path Planning and State Machine for the Rexarm

In this task, we designed a state machine to pick up a block present in-front of the of the Rexarm. To begin, the robot starts in a position where the arm is pointing straight up. The robot then calculates the inverse kinematics, and it sets the angles of the robot in order from gripper to base. This heuristic was chosen to reduce collisions with walls, as if the gripper is angled first, the robot can maneuver closer to walls without fear of the fully extended arm colliding into an object. However, this heuristic will only prevent collisions if they are preventable. In the future, the arm should consider SLAM data to determine if the arm is able to rotate to its desired positions.

Once the arm is in position, the gripper will close, and the arm will retreat into a L shape, where the gripper is pointing up. This is used as a waypoint to enter in to an S position, where the gripper is at a 90-degree angle. The waypoint is necessary, as the entire arm is too heavy for the base motor to lift on its own. The S shape is done to avoid any interaction between the lidar and the arm. From the S position, the robot can go to the home state (arm straight-up), without any additional waypoints. From the home state, the robot can go through the inverse kinematics to determine a drop-off position.

In terms of difficulties encountered during this task, there was two: the fact that our robot targeted the center of the closest tag and that the robot arm needed to be shifted the account for the gripper being in the exact center. In terms of the first problem, initially our robot targeted the center of the actual tag it saw, which means it would not actually attempt to grab the block; it would attempt to grab the tag. We shifted the coordinates arm attempted to grab by keeping the X and Z position but shifting the Y back. In terms of the second problem, we solved this by taking the unit vector of the X and Y positions and multiplying it by some small offset (in our case, we chose 0.02), and then adding this offset to the original location the gripper would land, effectively shifting it in the correct direction. Here is a demonstration of the functionality described in this section.

VII. Mobile Manipulation

In this part, we integrated the SLAM, Odometry and Rexarm pickup together. We initially faced multiple issues, such as our internal motion controller unable to spin the robot in place. Additionally, we had faced issues with the slow refresh rate of the SLAM pose in our Rexarm code, as the position the robot maneuvered to relied heavily on a correct SLAM pose. For example, when moving towards a block and picking it up, the SLAM pose was incorrectly received as an older SLAM pose. This would cause our robot to inaccurately move to the wrong position, and then fail to see a block when it attempted to pick up a block. This plagued our code base for a long time, and we even began to innovate solutions such as spinning around when we reached our position, and move closer to wherever we see the block.

Aside from picking up 1x1 blocks, the two corner cases that our mobile robot covers are picking up 3x1x1 blocks, and picking up blocks jammed into the corner. More details about the specific implementation are given in the description of the state machine.

VIII. Cleaning Up an Environment

A. High-level State Machine

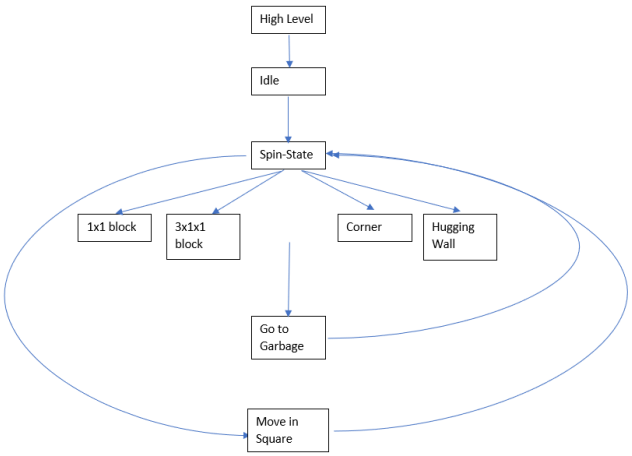

Fig. 3. State Machine

B. Spin State

To begin, the robot starts in spin-state. In this state, the robot spins 360 degrees and looks for a block in the area. If it

sees a block, it will proceed to 1x1 block, 3x1x1 block, corner or hugging wall state depending on the type of block it has seen. Otherwise, the robot will move to the next waypoint in a predefined square which fits to the environment it is attempting to explore and begin this process again (starting with spin-state).

C. 1×1 *Block State*

If the robot sees a 1x1 block, it will attempt to navigate towards it. However, we noticed that the angle of the robot after the navigation completes does not guarantee that the block will be seen by the camera once the robot arrives to its destination. Therefore, we do additional scans by rotating the robot dynamically once it reaches the target destination. First, the robot rotates between -10 and 10 degrees and determines if it sees the block. If it does, then we ensure our angle is turned towards the block and then run the code from Path Planning and State Machine for Rexarm. If it does not, then we increase the angle difference, and continue scanning until we find the block. Additionally, while we pick up the block, we will recognize its color and save it in the state machine for use later. Then, the robot will switch into the go to garbage state.

D. 3×1 *Block State*

The robot gripper is unable to directly pick up a $3 \times 1 \times 1$ block when the block is perpendicular to the forward direction of robot due to the limitation of gripper size. So we designed a new algorithm to pick up a $3 \times 1 \times 1$ block at any time.

The algorithm is composed of 3 parts. The first part is pose determination: If the robot sees more than 1 tag1, it will calculates the pose of the $3 \times 1 \times 1$ block in the world frame and decide whether it is pickable or not.

The second part is driving to the picking position: If the robot views the long block as not pickable, it will calculate a picking position based on the pose of the long block and drive towards that position.

The third part is spinning and pick: After reaching the picking position, the robot will start spinning around as we described before. As soon as it sees the block, it will use the same strategy of picking up 1x1 block to pick up the long block because it's guaranteed to be pickable.

Here is a Video Demo for picking up 3x1 block.

E. *Corner Block State*

The robot first maneuvers it's gripper into a position ahead of the robot but out of the way of the camera to avoid collisions with the wall later. Then the robot navigates toward the corner with the gripper in the forward position until it is a predefined distance from the block. The robot then relocates the block and begins the 'scraping' procedure in which it brushes the arm along the top of the block in order to scrape it out of the corner. This scraping procedure is accomplished by setting 10 waypoints very close together with equal x and z coordinates but decreasing y coordinates. Once the block is out of the corner, the robot raises the arm out of the way of the camera and relocates the block again. Finally, the robot picks the block up.

F. *Go to Garbage State*

The go to garbage state will force the robot to follow the set waypoints back to the original position. We know that this path will always be clear, as it the robot will have picked up any blocks in this path first. Once the robot reaches its original position, the robot can locate the garbage and supply area. The robot can decide on which to put it in by examining the variable set by the 1x1 block, 3x1 block or corner block states. Once the robot has dropped the block into the garbage, it will proceed to return to the last waypoint that the robot cleared, and then begin spin-state again. We do another spin-state, as there is a possibility there will be two or more blocks that can be seen by the robot in a specific waypoint.

G. *Move in Square State*

The move in square state will move the robot to the next available hard-coded waypoint. This only occurs when the robot spins but does not see a block. This means that it has cleared the area around this waypoint, and it can move forward to the next waypoint.

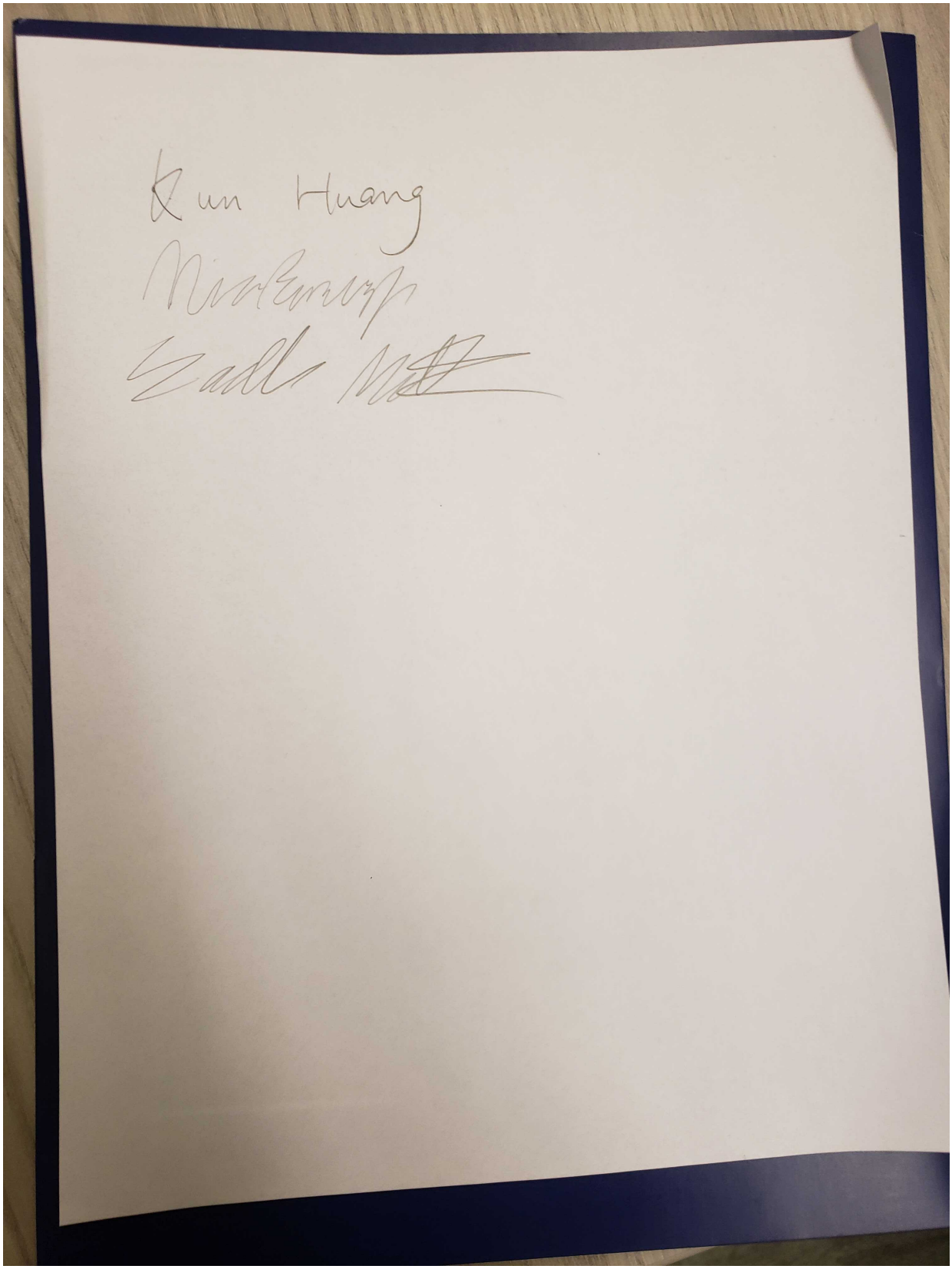

Fig. 4. Certification