Final Project Report

Bruke Baraki, Hongyi Lai, Leah Tesfa, Naomi Maranga, Yuka Nakada

I. Introduction

This report outlines our steps in completing the final project proposal, including the approach we used to write and optimize our code and evaluate our performance. Our approach utilized most of the methods and code developed in previous labs, as discussed later.

II. Method

1. Inverse Kinematics

Inverse Kinematics (IK) is the process of calculating the joint angles required for a robotic arm to achieve a specific position and orientation. In our implementation, the ik.inverse function from solveIK.py calculates the joint angles(q) required to achieve a particular pose, the distance (dis) and angle (ang) of the end effector's error, the status of the solutions' convergence (success), and the collection of all joint angles the solver iterated through to get its solution (rollout). The main function of the IK class is inverse, which takes a target pose and an initial joint angle configuration (seed) as inputs, as shown below. The ik.inverse function serves an integral role in motion planning since position inverse kinematics was used.

```
q, dis, angle, success, rollout = ik.inverse(target_pose, seed)
```

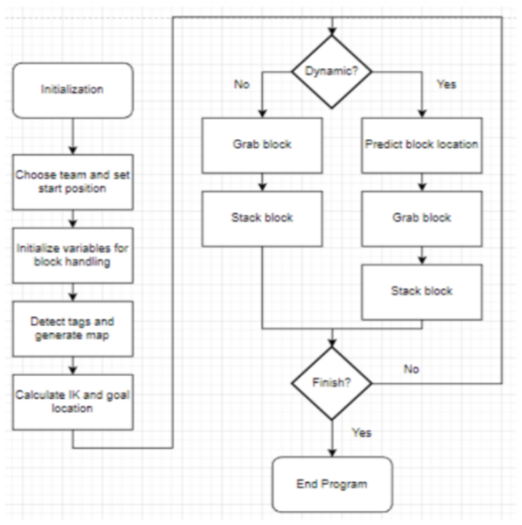

Initialization
Choose team and set start position
Initialize variables for block handling
Detect tags and generate map
Calculate IK and goal location
Dynamic?
No
Yes
Grab block
Stack block
Predict block location
Grab block
Stack block
Finish?
No
Yes
End Program

Figure 1: Flowchart for Methodology

2. Helper Functions

Helper functions were defined to simplify the coding process.

2.1. closestGrabPose(new_block_pose, hover_pose)

The end effector has 4 potential orientations while grabbing the static blocks, each of which are a 90^o rotation of the previous orientation. Although all 4 options can accomplish the goal of moving the static block to the goal platform, one option is the spatially most optimal. Furthermore, orientations that require a large change in joint angles are inefficient. The helper function closestGrabPose(new_block_pose, hover_pose) is tasked with finding the optimal joint angles. This is accomplished by post-multiplying the input "new_block_pose" by a rotation matrix that rotates the frame by 90^o about its z-axis. This is repeated 3 more times until there are four possible choices for joint angles. The change in joint angles was calculated by finding the difference between each choice for joint angles and that of hover_pose. The norm of the difference quantifies the amount of movement the robot arm has to make to get that orientation, as shown below.

```
q_choices[i,:] =  q - hover_pose # joint angles choices
q_change[i] = np.linalg.norm(q_choices[i,:]) # norm of the choice (smaller = better)
```

Obviously, a smaller norm is more efficient. By sorting each set of joint angles by their norm, the first element in the array is the most optimal choice, as shown below.

```
ind = np.argsort(q_change) #sort each choice by its norm
return choices[:,:,ind[0]], success
```

2.2. placeBlock(mode, numBlk, team):

Finding the correct pose to release the blocks in the goal platform takes multiple variables into account. The input “mode” was utilized since we are pursuing a 2 stack strategy, as opposed to having one main stack; thus, the x and y positions of the stack change based on which mode. The second input “numBlk” is responsible for raising the pose by a distance equal to the block’s length times the number of blocks already stacked in that mode, as shown in line 158.

goal_area[2,3] = numBlk*0.07 + goal_area[2,3]

Lastly, the input team defines which team the robot arm is part of since the poses differ, depending on the team.

2.3. transBlock(dyn_blocks)

Finding the translation portion expressed by x, y, and z positions of the first dynamic block found in detector.detections list. We first get the block’s pose and transform it into the end-efector’s frame since the pose is with respect to the camera’s frame. Then, we get the block’s pose in the world frame by considering the end-effector’s transformation. The output translation vector is simply the fourth column of such a matrix.

3. Strategy

3.1. Previous Method (Tested on Hardware)

The previous strategy that was tested on hardware was creating only one stack composed of the 4 static blocks initially and dynamic blocks after that, also placed on the same stack. The blocks were placed at each subsequent height step instead of being directly placed on the bench.

3.2. Current Method (Tested on Simulation)

The underlying strategy of block stacking is to create two locations for the blocks to be placed. This approach was utilized for its quicker runtime and lower toppling risk, as opposed to making one stack of blocks. The strategy consists of the following steps:

- Move the two static blocks closer to the goal area platform first
- Move the two static blocks further from the goal area on top of two blocks already stacked
- After placing block down, turn the end-effector’s joint 60^o in order to not push away from stack

3.3. Dynamic Block

3.3.1. Detecting Blocks

Our approach involves a while loop that only breaks when the camera’s detector detects blocks that satisfy constraints for picking up. We only wanted to consider the blocks that were incoming into the detector camera’s vision, and not outgoing or in the middle, as it would be too late for the robot to react and try to pick up the block given the robot’s reaction time.

Once a block is detected, the first element of the detector.detections list is checked and the x position of the block is analyzed. If the team is red, then the block must satisfy (block_x < 0.06) or (block_x < -10) in order to be considered a block incoming into the camera’s vision. If the team is blue, then the constraints are (block_x > -0.07) or (block_x > 10). If the constraints are not satisfied, then the detector’s detection field is emptied, and we need to continue the original loop.

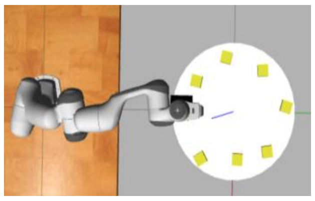


Figure #3: blue team robot in dynamic hover pose

Figure #4: block coming into vision

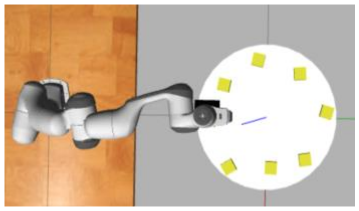


Figure #5: red team robot in dynamic hover pose

Figure #6: block coming into vision

3.3.2. Solving to Grasp Target Blocks

Once a block that satisfies constraints is detected, then our algorithm proceeds to calculate the distance at which the robot should reach the blocks. Our approach did not take into account the blocks' orientations when reaching for them, and thus the robot solves for the target's trajectory radius following the formula:

```
radius = np.sqrt(trans_block[0] ** 2 + (0.990 - abs(trans_block[1])) ** 2)
```

Where trans_block is the x, y, z positions for the target block. Once the radius is calculated, a transformation matrix is calculated and inverse kinematics is utilized so that the end-effector goes straight down to grasp the block according to that radius. Before moving to that pose, the robot delays for a few seconds (time was calibrated to be 2.5 seconds on the red side and 1 second on the blue side during robot lab) utilizing the time.sleep() method built into Python. Then, the robot intercepts the block when it passes under the end-effector.

3.3.3. Checking Gripper's State (Not Tested on Hardware)

In order to optimize the time taken for the dynamic blocks' approach, we used the arm.get_gripper_state() function to check the width between the gripper's ends. If the distance between them was less than or equal to 2 cm (smaller than the block's side length of 5 cm), then the robot should restart the detection search since no blocks were grasped. This would save us time since the end-effector would not move to the goal area without a block grasped between its jaws.

4. Motion Planning and Execution

Motion planning is relatively simple, focusing on moving the arm between the start position, hover position, object position, and goal position.

To calculate the joint angles required for these positions, the code uses the inverse kinematics library. Once the joint angles are obtained, the motion is executed using the ArmController interface: arm.safe_move_to_position(hover_pose).

The ArmController interface is responsible for controlling the arm's motion, and the safe_move_to_position function takes a set of joint angles as input and moves the arm to the desired position while ensuring that the motion is collision-free and within the arm's limits.

The approach described for controlling a robotic arm appears to be technically sound and reproducible, given that it incorporates fundamental concepts in robotics, such as inverse and forward kinematics, coordinate transformations, and motion planning. Utilizing the ROS framework and established libraries for calculations, such as numpy, IK, and FK, ensures that the code is based on well-tested and widely-accepted methodologies.

5. Pseudocode

```
# Import necessary libraries                    import robotic_arm_api
```

```
import object_detector
import transformations

# Initialize the environment, robot arm, object detector, and related parameters
env = initialize_environment()
robot_arm = robotic_arm_api.initialize_robot_arm()
object_detector = object_detector.initialize_object_detector()
team, goal_area, hover_target = initialize_team_and_targets()

#Move the robot arm to the starting position
robot_arm.move_to_start_position()

# Compute the transformation from the camera to the end effector frame (H_ee_camera)
H_ee_camera = compute_camera_to_end_effector_transform()

# Move the robot arm to hover over the static blocks
robot_arm.move_to_hover_position(hover_target['static_blocks'])

#Loop through the static blocks
For static_block in static_blocks:
    #Determine the block positions and orientations
    block_position, block_orientation = object_detector.detect_block_positions_and_orientations(static_block)

    # Compute the block's position in the end effector frame and global frame
    block_position_ee, block_position_global = transformations.compute_block_positions_in_frames(block_position, H_ee_camera)

    # Determine the best orientation for grasping the block
    grasp_orientation = compute_best_grasp_orientation(block_orientation)

    # Move the robot arm to align the end effector with the block
    robot_arm.align_end_effector_with_block(block_position_ee, grasp_orientation)

    # Execute the gripper command to grasp the block
    robot_arm.gripper_grasp()

    # Calculate the target position for placing the block
    target_position = compute_target_position(team, block_position_global)

    # Move the robot arm to the target position and release the block

    robot_arm.move_and_release_block(target_position)

    # Move the robot arm back to the hover position for static blocks

    robot_arm.move_to_hover_position(hover_target['static_blocks'])

# Loop through the dynamic blocks
for dynamic_block in dynamic_blocks:
    # Move the robot arm to hover over the dynamic blocks

    robot_arm.move_to_hover_position(hover_target['dynamic_blocks'])

    # Detect the dynamic blocks and filter out any false detections
    dynamic_block_position, dynamic_block_orientation = object_detector.detect_dynamic_blocks_and_filter(dynamic_block)

    # Compute the block's position in the end effector frame and global frame
    dynamic_block_position_ee, dynamic_block_position_global = transformations.compute_block_positions_in_frames(dynamic_block_position, H_ee_camera)
```

```
# Determine the best orientation for grasping the block
 grasp_orientation = compute_best_grasp_orientation(dynamic_block_orientation)

# Move the robot arm to align the end effector with the block

robot_arm.align_end_effector_with_block(dynamic_block_position_ee, grasp_orientation)

# Execute the gripper command to grasp the block
 robot_arm.gripper_grasp()

        # Calculate the target position for placing the block
         target_position = compute_target_position(team, dynamic_block_position_global)

        # Move the robot arm to the target position and release the block

        robot_arm.move_and_release_block(target_position)

        # Move the robot arm back to the hover position for dynamic blocks

robot_arm.move_to_hover_position(hover_target['dynamic_blocks']
```

III. Evaluation

Our evaluation was performed mainly in the simulation environment, as the robot lab times were very limited. The main evaluation metrics were:

- time to finish the process: how long did it take to complete the process?
- success/failure rates in grasping blocks: how many times did the robot successfully grasp and place the blocks at the correct orientation?
- number of blocks stacked at once: similar to success/failure rates, how many blocks were stacked at each trial?
- precision of our stacking (how straight our blocks were relative to each other): would there be a risk of the blocks tipping over in the real competition?

a. Simulation

In the simulation, our metrics were analyzed by timing the stacking process, taking the ROS realtime factor into account, as well as taking notes on how many times our robot successfully grasped the blocks in each simulation and placed them in the expected position. Those metrics and precision in stacking were all analyzed visually without any actual measurements taken in the simulation environment, although our initial idea was to print out the actual position of each block and see how straight the stack was.

Static Blocks

Upon conducting several tests for performance evaluation, we noticed that the static blocks were randomly generated at different orientations each time ROS was launched utilizing "roslaunch meam520_labs final.launch team:=red/blue" in the terminal. Utilizing this fact, we performed 5 experiment trials for each side (red and blue), each trial consisting of the robot's camera reading and solving positions for all 4 static blocks on the table. The tables below (Tables 1 and 2) consist of the data collected in our experiments.

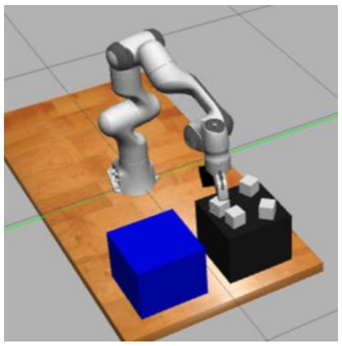
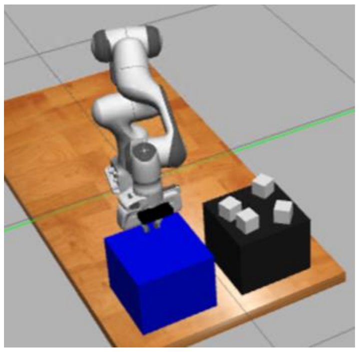
Time taken is expressed in minutes and seconds in the format min:sec and takes into account the time in simulation and the real time factor to express the time that would take in theoretical (ideal) hardware. Generally, the real time factor on ROS simulation varied between 0.45 to 0.7, and on average it was consistently 0.50 – this means that the simulation time was double the real-time, and that was the time listed on the table.

The success rate is how many blocks are successfully grasped and stacked on the goal table, and is expressed as a ratio n/4, where n is the number of blocks grasped.

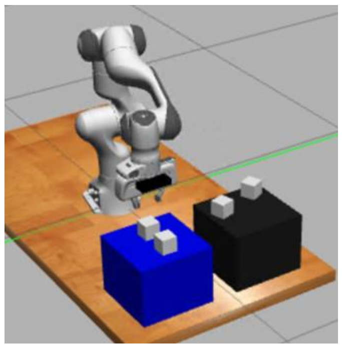
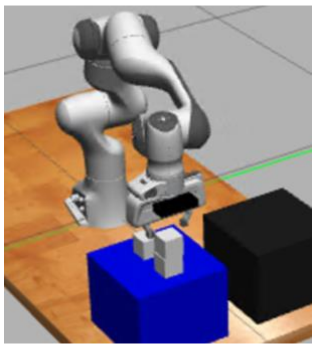
Blue Side		
Trial Number	Time Taken (min:sec)	Success Rate
Trial #1	2:33	4/4
Trial #2	2:02	3/4
Trial #3	2:26	3/4
Trial #4	2:15	4/4
Trial #5	2:14	4/4

Table 1: Data collected in simulation for the blue team

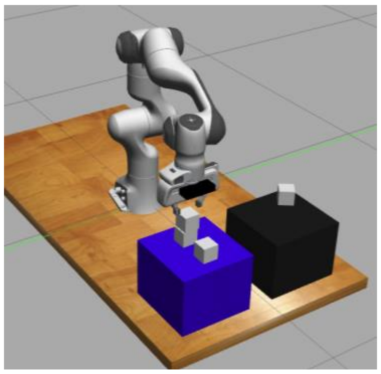
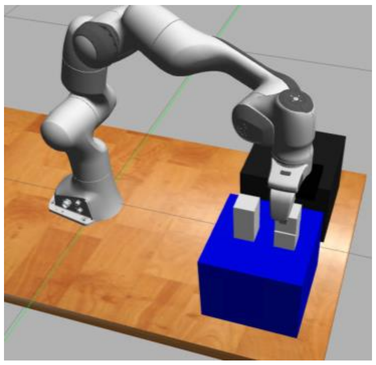
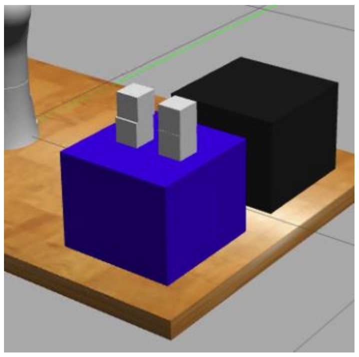
The robot had a 90% success rate on the blue side, and the average time to complete the process for static blocks was 2:18. On both trials in which the robot failed in grasping the block, it seemed to solve for the block's orientation incorrectly, either by exceeding joint limits for end-effector's rotation or location, and thus not grasping the blocks stably (Figure 7). As a result, the blocks slipped as the end-effector closed or were grasped at an angle different from the correct orientation (that is, the end-effector did not grasp the block at the center line) and the stack was not straight when the blocks were placed.

Figures #7 and #8: Block is grasped at incorrect orientation and slips (invisible block is placed)

Figures #9 and #10: robot proceeds to place the next blocks, but the one that slipped is pushed out

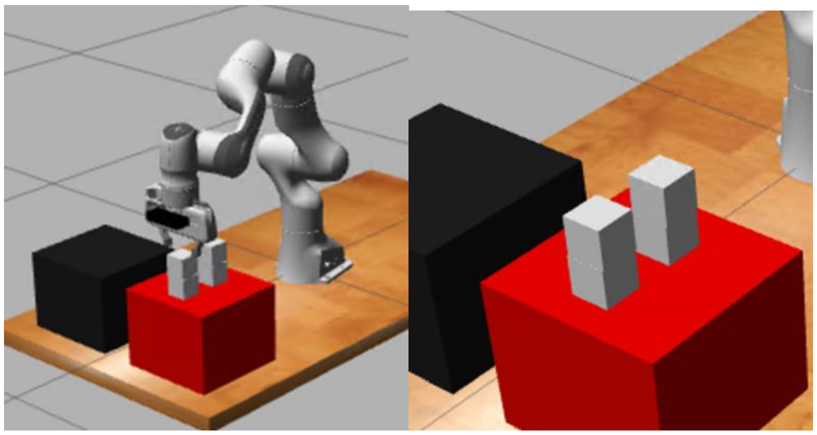




Figures #11, #12, and #13: blocks were successfully at slightly different angles than block orientations, and the stacks are not perfectly aligned

Red Side		
Trial Number	Time Taken (min:sec)	Success Rate
Trial #1	2:39	4/4
Trial #2	2:30	4/4
Trial #3	2:23	4/4
Trial #4	2:41	4/4
Trial #5	2:25	4/4

Table 2: Data collected in simulation from the red team

The robot had a 100% success rate on the red side, and the average time to complete the process for static blocks was 2:32. The end-effector seemed to reach the blocks at the correct orientation each time, and as a result the blocks also seemed to be precisely stacked on top of each other. As such, the red stack visually seemed more straight as compared to the blue side stack.



Figures #14 and #15: blocks are perfectly aligned in simulation on the red side

Dynamic Blocks

The robot arm successfully transitioned into the pose to hover over the dynamic blocks after finishing the process of reaching the blocks. In order to test the grasping of dynamic blocks, we let the robot detect and reach for the moving blocks at least 10 times and counted how many times it actually grasped the blocks. The time taken to reach the blocks and complete its trajectory to the goal position varied depending on the blocks' location, but the average for each trial was 1 minute and 13 seconds.

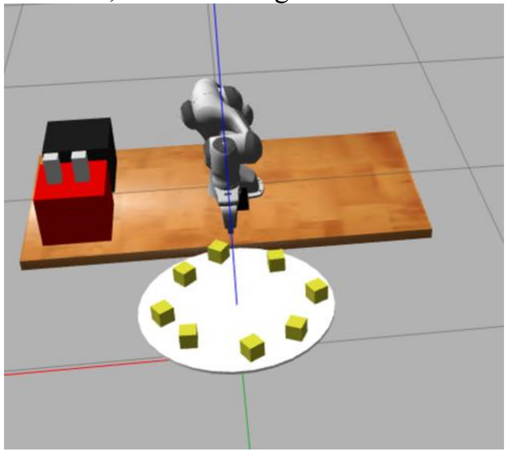


Figure #16: Robot successfully transitions into dynamic blocks' hovering position after stacking static blocks

The success rate of the red side was 5/10 = 50%, whereas the blue side was 6/10 = 60%. Some common failure cases were:

1. end-effector reached the block's future position too early or too late, not grasping it at the correct timing (Figure #17a);
2. end-effector "stabbing" the dynamic block as it reached the wrong orientation, reaching its top at 90 degrees (Figure #17b);
3. end-effector successfully finding the sides of the target block, but grasping its corners and letting the block slip as it's lifted (Figure #17c). Generally, grasping the blocks was a matter of luck and the stacks were not aligned, as the end-effector rarely grasped the blocks at the center line.

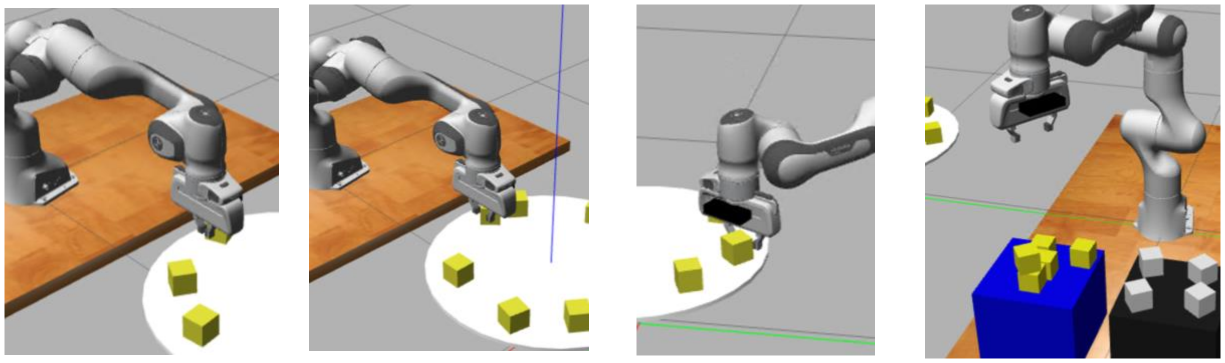


Figure #17: (a) Grasped too early and missed the block; (b) Grasped the corner and let it slip; (c) "Stabbed" the block from the top; (d) End-effector is rarely aligned with the block's center, and stacks are not straight as a result

Hardware

In order to perform thorough experiments to get the best improvements after testing on hardware, we recorded videos to check the differences between simulation and hardware performance. By taking videos, we were also able to keep track of the time taken to complete the process, how many blocks were successfully grasped and placed on the goal table, and visually check if the stacks were aligned.

Robot Lab

The testing was made before we changed our method for stacking, and thus the videos and pictures consist of a single stack of blocks. However, the general approach is very similar, and thus we consider the simulations and hardware testing to be comparable.

<u>*Static Blocks*</u>

The different tests were performed by asking the TAs to move and flip the blocks around each time we tested our process in order to create some randomness and eliminate bias. Our previous method was overall successful - being able to finish a whole stack with 4 blocks successfully once in 5 trials. In the other trials, the success rate was 3/4. The overall success rate was 4/5 = 80%, as the robot arm sometimes did not successfully grasp all blocks. However, as our approach dropped the blocks at a height, the stack was not completely aligned, and thus the precision was very low.

Generally, the robot did not collide with any of the blocks as the trials were performed. However, the end-effector would bump into some of the blocks after placing them before picking up the next block. This was a problem before updating our code to rotate the end-effector by 60 degrees after placing the block on the goal area.

The average time taken to complete the process was 2 minutes and 35 seconds.

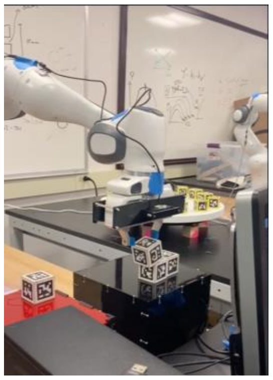


Figure #18: Tipping over

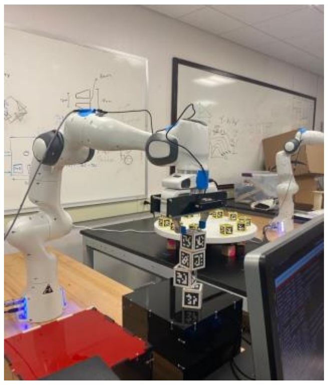


Figure #19: Successful stack

Video of the successful stack: https://youtube.com/shorts/qsHTTOV3kEs?feature=share

<u>*Dynamic Blocks*</u>

We were able to test the grasping of dynamic blocks on both red and blue sides, and we calibrated the time for which the robot would have to wait to solve for the block's position, as it was different for each side in simulation and hardware. Upon detecting the blocks, the robot was calibrated to delay its motion by 2.5 seconds on the red side and 1 second on the blue side. As in the simulation, this was tested by letting the robot reach for the blocks 10 times and checking the success rate and time. There were other teams testing the dynamic block picking at the same time, which added an element of randomness as the TAs replaced the blocks on the turntable.

Common failures observed were very similar to the ones presented in the simulation, with the robot reaching for the blocks too early or too late even after calibration. On the blue side, the "stabbing" collision mode seemed to happen more often than on the red side, likely due to the angled end-effector as discussed in the Analysis section.

Figure #20: robot reaches too early

Figure #21: robot reaches too late

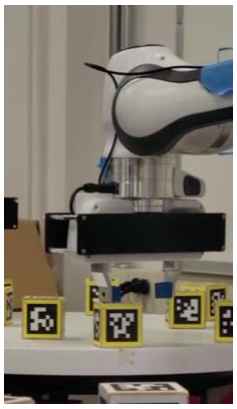


Figure #22: robot collides "stabbing"

The robot was able to grasp a few dynamic blocks, and the success rate was 4/10 = 40% on the red side and 2/10 = 20% on the blue side. Just like in the simulation, the stacks were not aligned when successfully placed on the goal area since the end-effector rarely grasped blocks along their center. The average time to complete each attempt of picking and placing the dynamic block was 30 seconds.

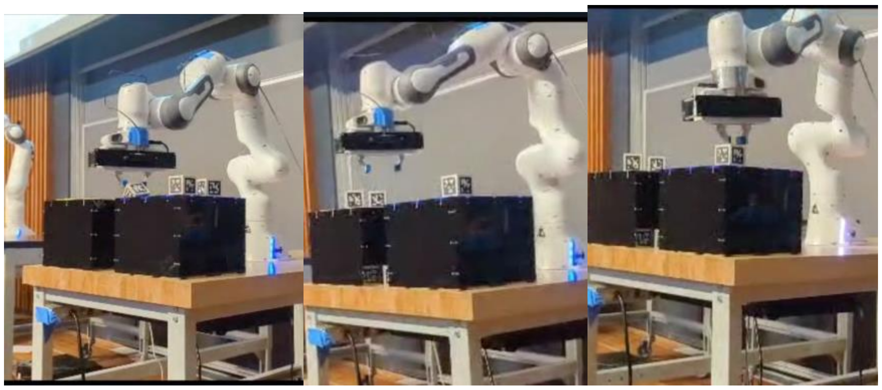
Figure #23: Stack was not aligned

Figure #24: Block was not grasped along center

Figure #25: As a result, the block was not placed aligned

Competition Day

On competition day, we were able to test our current approach on hardware (as listed on the method section) for the first time. On the blue team, our robot was able to grasp the first block but ended up dropping it before placing it in the goal area, likely due to the change of height after grasping that we made on our code. The robot was able to grasp and place the second and third static blocks, but it ended up colliding with the fourth block, likely due to noise in the camera detection. On the red team, our robot was stopped before grasping any blocks due to joint limits being violated. Unfortunately, we did not get far enough in the algorithm to be able to report any performance on dynamic blocks picking on the competition day.



Figures #26 a: Robot drops first static block; b: Successfully places second and third blocks; c: collides with fourth block

Figure #27: Robot software stopped before grasping

The time taken for the robot to pick and place the 3 blocks until it was software stopped was 1 minute and 30 seconds, which is about 30 seconds per block and would have been 2 minutes by the time it picked and placed the last block.

Competition day video (blue side): https://youtu.be/5F9duMfn8Ks
Competition day video (red side): https://youtu.be/06ccjTPcM3c

IV. Analysis

Analyzing our simulation and competition day results we studied the performance of these, and also discovered some of the possible reasons for any of the differences we observed between the performance of our static vs dynamic approaches, blue vs red performance, and hardware vs simulation results.

- *Differences in our blue vs red dynamic performance in simulation compared to hardware:*
  For dynamic blocks, we noticed that when on the blue team, our robot arm did slightly better than when on the red team in simulation. Referring to the images included in the Methods Dynamic blocks section, we illustrate how our robot arm on blue could see more in advance whereas for red it was directly horizontal, slightly obstructing the camera's field of vision. This greatly increased the general performance of our blue team as compared to the red team on simulation. However, in hardware, our red team surprisingly would perform better at stacking than the blue team. This may have been due to the fact that on the red side, it was easier to pick up our static blocks without interference due to joint limits. We noticed on competition day that our robot arm was stopped due to exceeding joint limits.

- *Changing method of stacking simulation vs hardware:*
  On the last week before the competition day, we were able to get our dynamic and static blocks working. We were also able to test these out on simulation and on hardware. However, in the hopes of increasing our task speed and eliminating fears of our stack toppling over, we tried to use a two-stack strategy. We were able to get successful results in simulation, but unfortunately due to the limitation of hardware time slots we were unable to test our new approach on hardware. Sadly, we used this version of the code on competition day without any hardware testing practice and our robot arm didn't perform as well as it had on simulation.

- *Dynamic Approach:*

Something we noticed about our dynamic approach is that our success strategy was based on the repeated trial and error of our robot arm to grasp the rotating blocks. In our method, we only considered the radius of the table to position our end-effector and did not take into account the implementation of orientation when trying to grab the dynamic blocks. This resulted in our robot arm often missing the moving blocks or colliding with them. Integrating block detection and adjustment of our robot arm with respect to orientation would have definitely greatly improved our dynamic block grasping success rate.

- *Static Approach:*
  Something we noticed on competition day was that when it came to static block stacking, some of the better-performing teams approached a static block, stopped, solved for orientation, and then approached the block to grasp it. In our method, we solved for orientation as we approached our block. On the day of testing, I believe also due to the slight change in the robot arm setup, using this approach resulted in collisions between our robot arm and the blocks.

- *Static Stacking:*
  When stacking our static blocks, our stacks were not fully aligned, and the reason for this was that our method dropped blocks at a set height away from where our last block was placed instead of placing them down on the table. We did this to avoid toppling our entire stack when stacking. We went with this approach since we were not being evaluated on perfect alignment or stability.

V. Lessons Learned

Some insights gained from completing the final project were:

- The importance of hardware testing before deploying a robotic solution. As we only tested our final solution on hardware during the competition, we were not sure what to expect. In the real world, we understood why it is so important to use hardware-tested solutions.
- The importance of analyzing and testing multiple approaches and not only a single solution. As we came and tested our solutions multiple times, we did not realize the possibility of solving for orientation before completely solving for the block's location. This resulted in a collision during the competition.
- The project would benefit from better time management by setting goals along the process and choosing times that work for all teammates using tools like When2Meet.
- The project would benefit from better and more frequent communication among team members
- The project quality would improve from better organization and task communication by using tools like GitHub and Trello to keep everyone in the loop without having to text or email each other.

If we could come back one semester and redo the project, we would have made sure to consider more cases in which the robot's end effector might be counted as in a collision state, not allowing us to compete. In addition, we would have asked the teaching staff further questions about joint limit violations and possible noises in the camera and how best to resolve this.

GitHub was used throughout the project, but since some of us were new to this tool, it was difficult to keep everyone in the loop when the code was updated. Having an additional tool such as Trello would centralize the information and help everyone stay updated.

This project has taught us a lot not only about robotics and the technicalities of working with robots but also about teamwork and the importance of communication.