MEAM 5200 Final Project Report

Avaniko Asokkumar, Vanessa Gong, Wilson Hu, Ashna Khemani

I. Introduction

The goal of the Pick and Place Challenge was to maximize our score within the given time constraint of 4 minutes. Points could be achieved by picking any number of static and dynamic blocks around the game field and placing them onto our goal-scoring platform. More points are given to the successful placement of dynamic blocks on the platform. Points were increased for blocks that were stacked on top of other blocks. Ultimately, the goal for our system was to stack as many blocks as possible on the scoring platform.

The challenge presented multiple technical obstacles that required our team to leverage concepts from throughout the course. We needed robust block detection capabilities, precise grasp planning to reliably pick up blocks, an effective stacking strategy to build a stable tower, and all these components needed to operate efficiently within the time constraint. Our solution prioritized reliability and speed over complexity, opting for a deterministic approach with pre-calculated positions and safety offsets. We determined that the constrained predictable environment didn't warrant the computational overhead of more complex approaches, developing separate strategies for static and dynamic blocks that accounted for their different behaviors and requirements.

This report details our methodology, implementation, and testing results, along with an analysis of our approach's strengths and weaknesses. We highlight key insights gained from the project, particularly regarding the differences between simulation and hardware implementation, and discuss lessons learned that could improve future robotic manipulation tasks.

II. Methods

We split our flow into two parts, stacking all static blocks in a single tower first, then adding as many dynamic blocks as possible. Static and dynamic scoring can both be broken down into two sub-phases: grasping (approach block, grasp, retract to clear table) and placing (move to scoring platform, approach, release block on tower, retract). We chose not to implement any obstacle avoidance path planning, like RRT or APF, in our solution as we determined that the tradeoff between the robustness of adding the algorithm versus the computational time cost (and to a lesser degree, implementation time/complexity) was not worth it. The environment of the robot was simple and consistent enough to navigate without any collisions via a few precalculated positions, repetitive motion, and hardcoded safety offsets.

The foundation of our solution was based on the forward kinematics and inverse kinematics algorithms that we had developed in class. Our solution also utilizes the ArmController object as well as the ObjectDetector object to translate our task solution into the real world. The ArmController object is used extensively throughout our code to move the robot arm to our desired positions that are calculated through our IK_position_null.inverse() method. The ObjectDetector is at the forefront of our solution's vision pipeline. Other technologies that are integral to the vision pipeline of our solution include the camera on top of the end effector and the AprilTags on the faces of the blocks. While we did not implement the pose detection algorithms, the camera uses the April tags to calibrate the 3D position and orientation relative to the camera.

Although our strategies for static and dynamic block stacking are very similar at a high level, our methodology for implementing each part had a few key differences.

Static Blocks

Our strategy for approaching the static block stacking was to keep the gripper inline with the world frame's z-axis during the entire stacking sequence. This meant that the gripper would approach the block directly from the top when grasping the block and stack the blocks from a top-down approach as well. We believed that this strategy would be the simplest way to consistently grasp and stack blocks without worrying about other parts of the robot arm colliding with the blocks. Since we weren't implementing any obstacle avoidance algorithms, we valued having a strategy that limited unpredictable positions for the robot arm.

The flow for the static block begins with scanning the game field for static blocks. We precalculated starting scanning positions for the robot depending on whether it was on the red or blue team:

Red Team Scanning Position: q1 = np.array([-pi/12, 0, 0, -pi/2, 0, pi/2, pi/4])

Blue Team Scanning Position: q1 = np.array([pi/12, 0, 0, -pi/2, 0, pi/2, -pi/4])

These positions were chosen based on the given coordinates of the goal platforms in the world frame as well as the position of the camera. In this position, the face of the camera lens is parallel with the platforms, allowing for optimal viewing of the top of the blocks and their AprilTags.

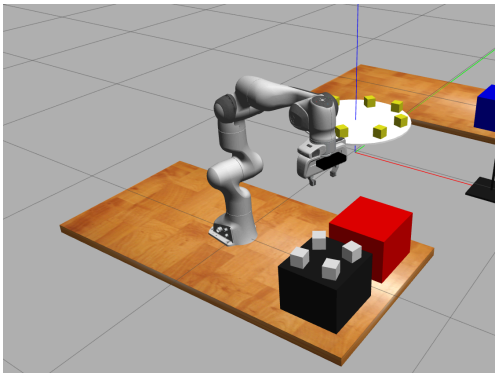

Figure 1: Scanning Position

The robot then iterates through each of the detected blocks and begins the grasping and stacking protocol. The grasping protocol begins with the arm moving to a calculated "pre-grasp" position, which is calculated by adding a 5 cm offset to the z-position of the homogeneous transformation matrix of the block in the world frame. We included this 5 cm offset because it would give us enough clearance to avoid hitting the block, since the block itself is 5 cm tall, which then sets the robot up to approach grasping the block by solely moving the gripper in the z-axis. Eliminating movement in the XY plane prevents the gripper from accidentally hitting or moving the block before fully grasping it. The robot then moves 2.5 cm down to grasp the block at its midpoint. If the block has been successfully grasped, then the robot moves on to stacking the block. If it has failed at grasping the block, the robot moves back up to the "pre-grasp" position and then moves on to attempting to grasp the next block in the list.

Once the block has been successfully grasped, it moves to the "pre-place" stacking position. In order to save computational time, we hardcoded each of the "pre-place" positions for the 4 blocks. We calculated the joint angles by setting the xy coordinates of the "pre-place" position based on the position of the goal

platform and adjusting the z coordinate position based on the block number in the stack. The x position is 0.562 m. If the team is blue, the y position is 0.1295 m. Otherwise, if the team is red, the y position is -0.1295 m. These xy coordinates correspond to being around the middle of the goal platform. The z position of the "pre-place" position is determined by this equation: 0.3 m+(0.05 m*n), where 0.3 m is the height of the goal platform which is 0.2 m plus a 0.1 m safety offset above the block set, 0.05 m is the height of a block, and *n* is the number of blocks that have already been stacked. We used the IK.inverse() function to get the corresponding joint angles for each of these positions. This resulted in the following "pre-place" positions, depending on whether we were the red or blue team:

	Block 1	Block 2	Block 3	Block 4
Red Team Pre-Placement Positions	Joint Angles = [0.16434038, 0.11224476, 0.06402013, -2.00165449, -0.00836924, 2.11365574, 1.01768036]	Joint Angles = [0.14472336, 0.07112529, 0.08382376, -1.92266006, -0.00652431, 1.9935285, 1.01641096]	Joint Angles = [0.1341005, 0.04792929, 0.09434329, -1.82304242, -0.00472427, 1.87075575, 1.01513017]	Joint Angles = [0.13225951, 0.04390705, 0.09647086, -1.70064968, -0.00429222, 1.74435129, 1.01477732]
Blue Team Pre-Placement Positions	Joint Angles: [-0.09338272, 0.11305076, -0.13721537, -2.00162768, 0.01802109, 2.11355297, 0.5463566]	Joint Angles: [-0.10158856, 0.07144918, -0.12809762, -1.92265161, 0.00999976, 1.99349868, 0.55193304]	Joint Angles: [-0.1061187 , 0.04807606, -0.12295617, -1.82304082, 0.00616943, 1.87074914, 0.55464103]	Joint Angles: [-0.10754422, 0.04402632, -0.12181595, -1.70064853, 0.00542955, 1.74434653, 0.5552173]

Table 1: Pre-placement joint angles for red and blue team

The gripper then moves directly down from the "pre-place" position to the "place" position. These are another set of hard-coded joint angles that we calculated using the same method as the "pre-place" joint angles. The only difference is that now we get rid of the 0.1 m safety offset in the z-axis, so the gripper no longer hovers above the goal platform or block tower, but instead lowers itself and places the block on top of the surface below it. The joint angles for each of the four blocks are as shown below:

	Block 1	Block 2	Block 3	Block 4
Red Team Placement Positions	Joint Angles = [0.21222456, 0.20547406, 0.01464937, -2.08327383,	Joint Angles = [0.17774189, 0.13922471, 0.05027531, -2.03373044, -0.00846151,	Joint Angles = [0.15337061, 0.08950842, 0.07515577, -1.96465935, -0.00757921,	Joint Angles = [0.13832065, 0.05722294, 0.09017392, -1.87552598, -0.0055065,

	-0.0039686, 2.28872255, 1.01457459]	2.17276478, 1.0177212]	2.05390462, 1.01714562]	1.93251192, 1.01569457]
Blue Team Placement Positions	Joint Angles: [-0.07387047, 0.20795045, -0.15731538, -2.0830998 , 0.04293367, 2.28809782, 0.5293153]	Joint Angles: [-0.08781069, 0.14044689, -0.14317539, -2.03367921, 0.02423248, 2.1725706 , 0.54208306]	Joint Angles: [-0.09796996, 0.09002338, -0.13217116, -1.96464501, 0.01337874, 2.05385062, 0.54957162]	Joint Angles: [-0.10428418, 0.05743049, -0.12502489, -1.87552035, 0.00765264, 1.93249387, 0.55358509]

Table 2: placement joint angles for red and blue team

If the system detects that the arm has successfully released the block onto the stack, the number of successfully stacked blocks is increased by one in the system memory. The program then moves the arm to another intermediary safe position that is 10 centimeters above the most recent stacked block position. This gives the arm clearance to move on to attempting to stack the next static block. If the system detects that the release of the block has failed, the system will move back to a predefined safe position defined by these joint angles: [0, 0, 0, -pi/2, 0, pi/2, pi/4]. The algorithm will then proceed to attempt to stack the next static block in the queue. This workflow repeats until there are no more static blocks left in the initial stored list of blocks.

Dynamic Blocks

Our strategy for approaching the dynamic block stacking was to use a predefined grasping position on the rotating table, with multiple grasp attempts to grasp the dynamic block, before following a similar pipeline as the static block stacking to place the grasped block onto the tower. Given the difficulty of localizing a moving block and predicting the future pose, we opted for a simple, experimentally tunable strategy. We believed this strategy would be the simplest way to consistently grasp and stack dynamic blocks without worrying about ghost blocks or pose miscalculations.

Our program first positions the robot end-effector ready to pick up dynamic blocks rotating into it, with the gripper at a 25-degree roll angle. The pipeline first calls the grasp_dynamic_block() method, where the robot attempts to grasp a set number of times (typically 4) before retrying, with a 5-second rest period in the open gripper position. This allows the turntable to rotate dynamic blocks into the gripper, after which it then attempts grasping and checks for a complete grasp (by calling the is_block_grasped() method). If it fails, it reopens and tries again the remaining set amount of times before resetting to a position above the table and restarting the process.

In the event of success, it uses the move_to_goal_platform_predefined(), where a set of pre-place and place positions is stored, different from those of the static blocks. These joint positions were calculated in a manner similar to the static blocks, using current stack height to calculate the intended z-coordinate, but with the 25-degree roll angle included. These pre-place and place joint configurations are listed in the code.

This allows the dynamic blocks to be placed flat on the surface of the tower, since they were originally grasped at the same roll angle (25 degrees).

In the main loop of the program, this happens after all static blocks have been stacked, by repeatedly calling grasp_dyanmic_block() and stack_dynamic_block () to stack as many dynamic blocks as possible.

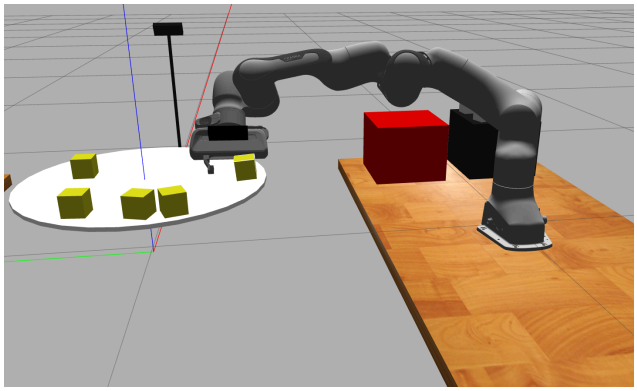

Figure 2: positioned to wait for and pick up dynamic blocks

Code Structure

Detecting Blocks — scan_for_block()

This is a method that encompasses the full flow of finding blocks on the game field. The inputs are an ArmDetector object, an ObjectDetector object, and a list of joint position vectors called scanning_positions. The method iterates through the scanning positions given, moving to each of them using the arm.safe_move_to_position() function, and uses the detector.get_detections() function to check if there are blocks within view of the camera. The loop continues to the next scanning position if no blocks are detected. If there are blocks within view, the robot stops scanning and transforms the positions of the detected blocks into the world frame using a homogeneous transformation matrix from the camera's FOV to the end effector's (ee) FOV is obtained from the detector.get_H_ee_camera() function. The transformation matrix from the world to the end effector is obtained from the forward kinematics function FK.forward(). All these transformations are applied to get the final block transformation matrix:

$$H^{block}_{world\ real} = H^{world\ ideal}_{world\ real} \ @\ H^{ee}_{world\ ideal} \ @\ H^{camera}_{ee} \ @\ H^{block}_{camera}$$

Correction term

Calculated ee position w.r.t. world frame (`FK.forward()`)

Camera position w.r.t. ee (`get_H_ee_camera()`)

Raw camera reading (`get_detections()`)

This transformation matrix is then passed through the helper function calc_target_pose() to obtain the reoriented matrix. Finally, each block is sorted into separate lists depending on if they are a static or dynamic block.

1. calc_target_pose() — helper function that redefines a block's z-axis to be aligned with the world frame

    a. This function takes in a homogeneous transformation matrix corresponding to the block's position and orientation from the world frame and realigns the axes so that the z-axis is in the same direction as the world frame's z-axis. This is accomplished by iterating through each of the first 3 columns of the matrix, taking the 3 three elements—since those are the elements that correspond to the rotation matrix between the block's coordinate frame and the world's coordinate frame–and then dotting this 3x1 vector by the z unit vector [0, 0, 1]. The vector corresponding to the dot product with the largest absolute value is stored as the new z-axis of the block's coordinate frames. If the dot product happens to be negative, then the vector is multiplied by -1 to make sure that it points in the correct direction. We repeat this process to find the appropriate reoriented y-axis of the block as well. We then calculate the new x-axis of the block by taking a cross product: $\hat{x} = \hat{y} \times \hat{z}$. This ensures that the resulting coordinate frame for the blocks is handed.
2. cam_offset() — helper function that is used only for hardware testing to help calibrate the camera
    a. Takes in three variables, x, y, and z, that correspond to offsets for each axis in the world coordinate frame. The function outputs a homogenous matrix ($H^{world\ ideal}_{world\ real}$) that can be applied to another transformation to get the appropriate offsets.

Stacking a Static Block — stack_block()

This is a method that encompasses the full flow of grasping and placing a static block on the current stacked tower. In the main loop, we iteratively call this function on each target block pose.

1. grasp_block() — safely grabbing a block from the static platform
    a. Move to a "pre-grasp" position that is 10cm above (in positive z0) the target block position
    b. Approach the block and close the gripper. Reattempt if the lock is not detected
2. inter_move() — retract the end effector 10cm in positive z0 to clear the static platform and avoid hitting other blocks
3. move_to_goal_platform() — safely depositing a block on the scoring platform
    a. Move to the appropriate hard-coded pre-place joint configuration (defined by how many blocks have already been stacked)
    b. Lower to the hard-coded placement position, and open the gripper
    c. Retract back to the pre-place position to avoid hitting stacked blocks

Stacking a Dynamic Block — 2 method loop

This part of the pipeline cycles between the following two methods in charge of grasping and stacking/placing the dynamic block. In the main function we set the number of iterations we loop this for.

1. grasp_dynamic_block()
    a. Create a transformation matrix for a pose that waits for and attempts to grasp dynamic blocks as they arrive (as shown in Figure 2 above). This contains the roll angle's rotation matrix, and an XYZ position of a point where dynamic blocks should pass.
    b. To find the joint angles for this pose, we use IK.inverse(), with the seed set to a configuration q_hover that corresponds to hovering over the dynamic platform. Having

the starting seed closer to the end configuration (compared to the last known position, which will be further away at the scoring platform) allows the IK solver to run faster.

c. Move the robot to the resulting joint angles. Repeat the following for 4 attempts:
    i. Open the gripper and wait a few seconds
    ii. Close the gripper. Check if a block is being grasped using the force sensor.
    iii. Try again if no block is detected. If a block is detected and successfully grasped, exit this loop and retract to the q_hover position.

B. stack_dynamic_block()
    a. From the q_hover position, we first move to the predefined, hard-coded pre-place position using move_to_goal_platform_predefined(). Then move to the place position (also hard-coded) and open the gripper. For the dynamic blocks, the hard-coded positions we calculated account for the fact that we grip the block from an angle, rather than simply from above like the static blocks; the desired pose encodes the appropriate rotation matrix in the correct orientation. This helps avoid contacting the tower from an angle, potentially causing it to get knocked over.
    b. Retract 10cm in positive z0 (upwards) to avoid hitting scored blocks.

III. Evaluation

Success Metrics

Our evaluation of our solution was split into two parts: simulation testing and hardware testing. Each was further split into two separate parts to evaluate static block scoring and dynamic block scoring. The metrics we used to evaluate the success of our tests were chosen based on the requirements of the game. The first quantitative metric we used to evaluate success was how many of each block we were able to place on the goal platform and how tall the resulting stack was. This metric can be summarized by the scoring metric provided by the game rules outline: points = value x altitude, where the value of the block is 10 points if it is static and 20 points if it is dynamic, and the altitude is defined as the distance between the center of the block and the goal platform. The second metric of success we used was the program's runtime. In the actual challenge, we only have 4 minutes to get the highest score possible, thus, we want to not only be able to stack as many blocks as possible, but do it as quickly as possible. Neither of these quantitative metrics needed targeted testing designed to evaluate them, and were something we took note of during each of our simulations and hardware testing.

In terms of qualitative success metrics, we wanted to evaluate the robustness, stability, and adaptability of our solution. To give more structure to these metrics, we developed guiding questions, highlighted in Table 3, for each of them.

Robustness	Stability	Adaptability
● Is it prone to getting stuck in loops/inactive states?	● Is the motion of the robot smooth/controlled? ● Are there any erratic movements or constant sensitive corrections?	● Can it handle blocks placed close together? ● Can it quickly adjust after bumping into a block? ● Will it avoid knocking

		over existing tower structures?

Table 3: success metrics guiding questions

To test robustness and adaptability in simulation, but more so in hardware, we would place the static blocks in various angle orientations as shown below in Figure 3. We also placed blocks in different positions spread around the goal platform to test edge cases of blocks on the edge of the platform and blocks being placed closer to one another.

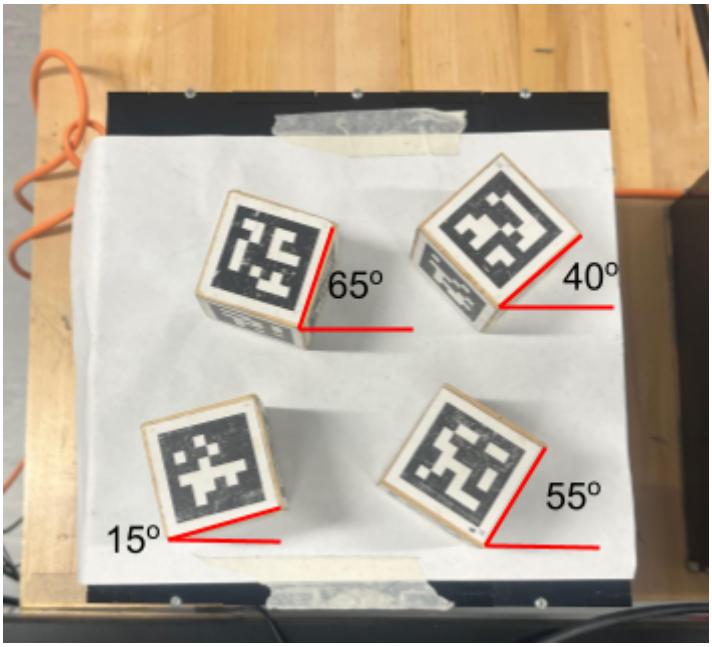

Figure 3: block orientations to test algorithm robustness

In terms of stability, this was something that we monitored mainly through observation. Some quantitative metrics that we took into consideration for stability were how tall the tower we were able to stack.

Static Block Testing Results

Using our success metrics, we went through several iterations of static block algorithms with minor tweaks to optimize block grasping success rate and runtime. Overall, we had 3 main iterations that we extensively tested on simulation and hardware. All three iterations had high success rates for stacking 4 blocks on top of each other in simulation for both red and blue team stacking trials, correlating to a score of 4000 points. During hardware testing, however, is where we noticed discrepancies.

For the first iteration, we tested it as a blue team robot on hardware. There was an error within our block axis alignment algorithm that led to the gripper improperly turning for blocks angled at 15-25 degrees. This led to a few of our test runs only being able to successfully stack 2 out of 4 blocks, corresponding to a 50% grasping success rate.

During our second iteration, we also signed up to test as a blue team robot. We addressed this issue so that the gripper would always be able to properly align with the orientation of the static block. We had greater

success with this method as it was able to consistently detect 3 out of 4 on the block table and stack them on top of each other, resulting in a grasping success rate of 75%. The average runtime of these 3 trials, where 3 blocks were successfully stacked, was 3 minutes and 28 seconds.

Our third iteration and final iteration aimed to cut down on runtime. We accomplished this by cutting out any unnecessary use of the IK_position_null.inverse() algorithm, as we observed that computational time between movements was quite significant. We ended up hard-coding many intermediary positions as outlined in our methodology. During our hardware testing sessions with this new algorithm, we ended up running into an issue where, in simulation, everything worked perfectly, but on hardware, our IK_position_null.inverse() function would continuously fail to find solutions to our outlined grasping positions. After 2 hardware testing sessions of troubleshooting, where one session was on the blue team robot 1 and the other session was on the red team robot 2, we were only able to fix this problem during our last session by adjusting the angular tolerance for the inverse kinematics end condition. Unfortunately, this meant that our only real data point for hardware testing came from our competition run. During our second round of competition day, we were able to successfully stack 3 static blocks within 2 minutes and 51 seconds. This is 82.2% less time than our previous iteration.

Dynamic Block Testing Results

We were able to test our dynamic block grasping a few times during our final testing session before the competition. A major problem we faced in simulation was that blocks had to be placed into the grippers while on the rotating table, or else they wouldn't slide into place, since the friction between roundtable and block was quite high in Gazebo. Based on our physical intuition, we guessed that this problem would go away on hardware. After adjusting the end-effector z-position for the hardware test, we were able to get 2 successful block grasps in a row, but we didn't have the stacking program configured to place these blocks at the right angle at the time. This was okay, as the main assumption that we had proved was that the blocks would be able to slide into the open gripper in real life, even if it didn't work in simulation.

We performed the rest of the testing in simulation, and given the similarity in block stacking between static and dynamic blocks, we were able to have successful runs of stacking at least 2 dynamic blocks on top of static blocks in simulation. With blocks above 2, we ran into some joint limit issues that we didn't get a chance to resolve, even after trying different predefined joint configurations, with reseeded IK solver computations.

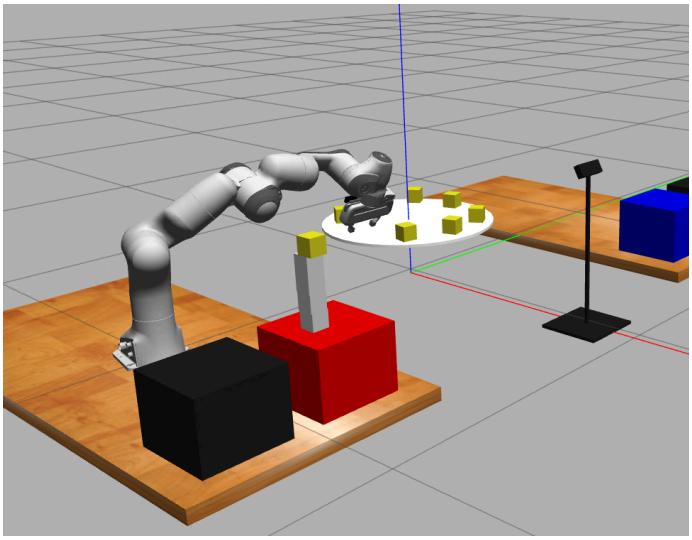

Figure 4: successful dynamic block place after static stacking

IV. Analysis

Our strategy going into the competition was to stack all four of the static blocks first and then attempt to stack the dynamic blocks on top. We were also aiming to disrupt the stacking procedure of the other team via our dynamic block grasping algorithm by either knocking a majority blocks off the dynamic table, or moving them so much or displacing them so the other team would not be able to get the blocks.

Static Block Analysis

During competition day, our solution worked properly for one of the rounds. During our first round of competition, competing as the red team, our robot was unable to successfully grasp any of the blocks because of an error with our camera calibration offset. This led to the gripper lining up properly with the block orientations right above the blocks, but attempting to grasp each block right above the proper grasping location. For the second round, we switched sides to the blue team and were more successful as the calibration was correct. Our solution attempted to stack all 4 blocks, and was able to successfully stack 3 out of four, for a 75% grasp success rate. The entire process took 2 minutes and 51 seconds.

Given our evaluation metrics, we would conclude that our solution was not completely successful. The robustness and adaptability of our algorithm could have been improved upon, as during hardware tests, our robot was only able to stack 3 blocks most of the time before the sequence ended and moved on to dynamic stacking. For all of these successful 3-block stacked hardware test runs, however, the robot only attempted to grasp 3 out of the 4 blocks in total, not all 4. Thus, we can attribute this failure to stack all 4 blocks to limitations of our detection algorithm. It is likely that our scanning positions were unable to capture all 4 blocks in the camera's FOV, thus only adding 3 out of 4 of the blocks to the grasping and stacking sequence each time.

Another thing worth noting is that we initially intended to make our algorithm more robust by adding a feedback loop to check for ghost blocks, but we had only ever encountered ghost blocks during our first-ever hardware testing system. We concluded that the extra time it took to add in those extra scanning poses was not necessary for our solution to be successful.

Dynamic Block Analysis

During competition day, our solution for dynamic stacking faced several challenges that limited its effectiveness. In our first round, our dynamic solution was correctly positioned but was unable to successfully grasp any blocks due to an implementation issue: the number of grasp attempts in our algorithm was set too low (only attempting 4 grasps before resetting). With the relatively slow rotation speed of the turntable and the unpredictable positioning of dynamic blocks, this parameter setting proved to be insufficient. The robot would wait in position, attempt grasping a few times, then reset its position without having had an adequate opportunity to encounter and grasp a dynamic block as it passed by.

For the second round, we encountered a different issue related to hardware calibration. The predetermined gripper position offset was set too low in the z-axis, causing the robot arm to make slight contact with the rotating table. This minor collision was enough to impede the table's rotation, effectively stopping it from moving. This issue not only prevented our robot from grasping dynamic blocks but also inadvertently disrupted our opponent's ability to grasp dynamic blocks since the table was no longer rotating properly. While this unintentional interference with the opponent's strategy might have been beneficial from a competitive standpoint, it prevented us from testing our full dynamic block stacking capability.

These issues highlight the challenges of parameterizing robotic interactions with dynamic objects. The simulation environment did not adequately capture the physical constraints of the real system, leading to discrepancies when porting our solution to hardware. Our dynamic block solution required more careful tuning of parameters than the static one. Unlike static block manipulation, which could rely on precise pose information, dynamic block manipulation proved much more sensitive to small implementation details and physical offsets.

For future iterations, we would implement adaptive parameters for dynamic grasping that can be more quickly tuned during competition, such as for the number of grasp attempts and the height offset.

V. Lessons Learned

This project was a valuable experience in integrating key concepts from class in an organized and structured manner. Working in both simulation and a physical environment with a task that had more robot-world interactions highlighted many of the differences between the real world and simulation, and areas where one seemed “better” than the other. Inconsistencies with the hardware, like offsets or the camera FOV being different, were much more pronounced, and we had to figure out how to adjust for them in a way that was logical and reliable. However, we noticed the simulation did not model friction as consistently, and had a lower performance at the dynamic task than when we ran our code on the real robot. Ultimately, our system worked very well and reliably in simulation, but not on the physical robot. Our team did not plan for having enough time to adjust for some of the inconsistencies and variabilities of the real world, which resulted in only a semi-successful performance.

Another lesson we learned was the importance of version control, documentation, and organization. Our progress throughout the project was not linear; we would sometimes revert to older strategies or combine them with something we were currently testing. However, in moving back and forth between versions, many of the intermediate improvements we made would get lost in the shuffle. This resulted in having to reimplement them from memory or dig through many versions of the code. Through a more organized approach, we likely could have saved a lot of time and had a more complete implementation that would work both on simulation and the physical robot. With time more meaningfully spent, we also would have

liked to create more detailed testing plans for our limited lab slots. Rather than aiming to get a particular part of the code working, we would have focused more on tuning our system to account for the simulation vs.-real discrepancies, exploring edge cases of our implementation, identifying underlying errors like our IK convergence issue ahead of time, or even experimenting with new parameters like the robot's speed setting.

Overall, the more complex nature of this task made the unideal, imperfect nature of physical vs. simulated systems much more pronounced and relevant than the past labs we have completed. We needed to have a much more organized and planned-out approach to the problem, something we realized towards the end of the project window. However, we are confident that we were on track to build a fast and reliable system, and learned a lot about the development process along the way.