MEAM 520 Final Report

Andreas Alexandrou, Zachary Gong, Jason Friedman, Hunter Liu

December 2021

0.1 Method

First off, we wanted to structure our approach based how we wanted to attack the game. We obviously wanted to stack all of the static blocks, and we confidently believed we could stack each block white side up. The issue came with the dynamic blocks. Without the vision system, we were very unsure of how to blindly grab a block. We brainstormed ideas of moving the arm into the path of the dynamic blocks, and closing the gripper until we detect we were holding a block. However, this method seemed very random and prone to errors. But without the vision system, we could not think of a great way to pick up the dynamic blocks in a fast and efficient manner. Therefore, we focused on stacking the static blocks all white side up first, to maximize the amount of points we knew we could score, and then if we had time leftover attack the dynamic blocks.

Our approach can be broken down into 5 components: 1. block detection 2. pick up planning 3. pick up block 4. place block 5. rotate block if necessary. One of the first decisions we made was determining which partner's set of lib files we would use. After comparing some of our average run times from previous lab reports and discussing the the general approach that we each used, we determined that the speed and overall structure was similar enough to be interchangeable.

Block detection takes in the transformations of the tags in the camera frame from $detector.get\_detections()$ and returns a list of transformations of the blocks in the robot frame. The list is ordered based on the distance of the center of the block and the robot base.

Pick up planning uses a single block transformation in the robot frame to return 2 items: 1. list of transformations of the end effector (EE) in the robot frame that are possible positions and orientations that can pick up the given block and 2. list of corresponding end effector positions and orientations in the block frame to be used later in determining how to rotate the block such that the goal face is pointing in the positive world axis.

0.1.1 Block detection

We are given a list of tuples $(tag\_name, T^{camera}_{tagi})$ from $detector.get\_detections()$ of size 10 where tag_name is the tag labeled from 0 to 12. $tag0$ is the static tag, tags 1-6 are faces on the static block, and tags 7-12 are faces on the dynamic blocks. Assume that $tag0$ will be included along with the 4 tags corresponding to red or blue static team's blocks depending on what team we are assigned.

To get the list of ordered block transformations in the robot frame, we use three main functions: 1. *getTags* 2. *getBlocks* and 3. *getOrderedBlocks*.

getTags takes in as a parameter the list of tuples $(tag\_name, T^{camera}_{tagi})$ and returns the list of tuples $(tag\_name, T^{robot}_{tag_i})$ of size 4 where each index is the transformation of a tag on a static block in the robot frame.

$$T^{robot}_{tagi} = T^{robot}_{tag0} T^{tag0}_{camera} T^{camera}_{tagi}$$

T^{robot}_{tag0} is constant and equal to

$$\begin{bmatrix} 1 & 0 & 0 & -0.5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where the -0.5 in the first row, fourth column of T^{robot}_{tag0} is the robot x-axis translation to the $tag0$ position. We are given $T^{tag0}_{camera} = (T^{camera}_{tag0})^{-1}$ and each T^{camera}_{tagi}.

One incorrect assumption we initially made was that "tag0" would be the first tag in the input list of tuples which caused errors in simulation. Because T^{camera}_{tag0} is needed to calculate the desired T^{robot}_{tagi}, we first loop through the input list of tuples and check if the tag_name is "tag0". When it is, we calculate the inverse

to find T_{camera}^{tag0} and break out of the loop.

Next, we create a set where each element in the set is a string of one of the possible static block tag names "tag1", "tag2", ... "tag6". We iterate over the input list of tuples again and check if the *tag_name* is in the set of possible static block tag names. When it is, we calculate T_{tagi}^{robot} using the previous formula described and add that to the new list of tuples $(tag\_name, T_{tag_i}^{robot})$ which is returned at the end of the function.

getBlocks takes in as a parameter the list of tuples $(tag\_name, T_{tag_i}^{robot})$ and returns a list of size 4 where each index, T_{block}^{robot}, is the 4x4 transformation of the static block in the robot frame.

$$T_{block}^{robot} = T_{tagi}^{robot} T_{block}^{tagi}$$

We can get each T_{tagi}^{robot} from the parameter, so we need T_{block}^{tagi}. T_{tagi}^{block} always stays the same according to the given diagram in Figure 2. of the final project instructions. We can use $T_{block}^{tagi} = (T_{tagi}^{block})^{-1}$. From lecture, we learned that the columns of the transformation matrix from coordinate frame A to coordinate frame B, T_A^B, are the coordinate basis vectors of B expressed in the coordinate basis vectors of A:

$$T_A^B = [[b_x]_A [b_y]_A [b_z]_A]$$

As an example, we'll explain the calculation of T_{tag1}^{block} to demonstrate how we created a dictionary with keys as the *tag_name* and values as the T_{tagi}^{block}. Note that this dictionary contained the key value pairs, $(tag\_name, T_{tag_i}^{block})$, for each of the 6 static tag names.

For T_{tag1}^{block}, the x-axis of "tag1" points in the block's y-axis, the y-axis of "tag1" points in the block's z-axis, the z-axis of "tag1" points in the block's x-axis, and "tag1" is 0.025 m, or half the width of the block, from the block's center on the block's x-axis, so

$$T_{tag1}^{block} = \begin{bmatrix} 0 & 0 & 1 & 0.025 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To return the list of 4x4 transformations of the static block in the robot frame, we loop through the list of tuples input and get the T_{tagi}^{block} from the dictionary using the *tag_name* from the first index in the tuple. After inverting T_{tagi}^{block} and pre-multiplying by t T_{tagi}^{robot}, we added T_{block}^{robot} to a list and returned that list at the end of the loop.

getOrderedBlocks takes in as a parameter the list of 4x4 transformations of the static block in the robot frame and returns the same list ordered with the first index as the block closest to the robot base and the last index as the block farthest from the robot base.

To do so, this function uses the built in Python *sorted* function that can sort a list based on another list of parameters. We input into the *sorted* function a list of tuples, $(block\_distance, T_{block}^{robot})$. The block's distance from the robot base was calculated by taking the first three rows of the last column in T_{block}^{robot} and calculating the norm of that vector.

0.1.2 Pick up planning

Planning how to optimally grip each block was a very complicated decision, which continually evolved throughout the course of this project. We strove to place all the blocks white side up, while operating as quickly as possible (constrained by 3 minutes in total), and avoiding collisions (with other blocks or the platform) at all costs. As such, much consideration was given to orientation of gripper upon block pickup and release, and the complexity of motion to reach each. Regarding gripper orientation, the top grips (approaching from the positive z) were preferable because they offered collision avoidance, compared to side grips (approaching in the world x or y) which might obstruct with an adjacent block or with the table.

Breaking it down, we observe that blocks can appear in three main orientations, each describing the orientation of the white side relative to the world frame: UP (1/6 cases); SIDE (4/6 cases); and DOWN (1/6 cases), each of which must be handled by our logic. The UP case was first handled by gripping from the

side (namely, whichever side was closest to the gripper base) to leave the +z white side exposed, but then we realized that a more optimal solution exists: Simply both pickup and release the block from overhead. The SIDE and DOWN cases however pose the challenge of required reorientation for Side Bonus points. We reasoned that attacking this problem would require rotating the gripper's EE orientation by 90 degrees between pickup and release. We narrowed down this task into two main solution options, casing on whether the translation and rotation happen sequentially (SEQ) or simultaneously (SIM).

We first explored the SIM option, benefit being speed and efficiency since only one total motion is required rather than two. Since nothing matters after the first three minutes, we chose to explore this option in the hopes that the quickness of a single block transformation lets us stack as many as possible. To allow for this, the gripper's pickup and release motions must include (in no particular order) one side action and one top action, with the rotation occurring in between during translation across the game board. One simplifying decision was to implement top grip pickups under any circumstances, uniquely avoiding unwanted collisions with adjacent blocks on the starting platform, yet as a byproduct leaving the inevitable side action to occur during block release. Unfortunately, early simulations showed that this cost of dangerous side releases was too large, as collision often occurred between the EE and target platform for the very first (thus least height) block. As such, we pivoted to the more conservative yet slower option of including rotation as a separate motion, executing only after the block's translation onto the top of the stack. Implementation and execution of each rotation is detailed below in "Rotate block" section.

Software implementation of this "Pick up planning" section occurred in our *getPickUpTransformations()* function, which takes in a single output T_{block}^{robot} from *getOrderedBlocks()*, and outputs a list of acceptable transformations T_{EE}^{block} and T_{EE}^{robot}, the latter which follows trivially using the argued T_{block}^{robot} to transform. Computing T_{EE}^{block} was a messy linear algebra process, but key steps are as follows. For one, we observed that whichever easily computed (using T_{block}^{robot}) block frame direction maps with the world -z, was the desired direction for the EE +z for top grip, which informed a whole column of our return matrix. Additionally, the EE +x had to map with one of the block's 4 remaining side directions, which informs another column of the return matrix. The remaining column could be determined using cross products, as rotation matrices are constrained to transform from one orthonormal basis to another, and must have determinant equal to 1.

0.1.3 Pick up block

Our approach to picking up the block is fairly simple. We are given a list of desired robot end effector orientations which we can pick up the block in. Initially, we were going to use RRT to guide our movement in the workspace. We would form an RRT between our current configuration to the goal configuration, which we can easily find by running the IK for the given list of end effector frames using the neutral configuration seed. We can then create another RRT from the current location of the block to the desired location to place the block. However, we ran into significant issues with the RRT. To begin, our RRT is not guided at all; nodes are randomly selected across the workspace. This means that the path is almost never the fastest path, and the robot moves seemingly randomly across the workspace on the way to the goal. This obviously wastes a lot of time, which is a big concern for this project. Additionally, to run the RRT we need to have a map of the workspace. Because blocks are constantly moving around over the course of the game, we would need to continuously update the map. And generally, RRT can take a significant amount of computing time. Because of all of these reasons, we decided to move away from the RRT method and simply use a waypoint method.

We wrote a *pickUpBlock()* function that takes in the list of desired end effector frames for the robot. It calculates the waypoints as locations 0.075m above the desired end effector frames, so 0.075m above the location of the block, in the same orientation as the desired end effector frame passed in. Looking at the workspace, we know that these locations will generally be clear - moving from a given configuration to these configurations will not collide with any blocks in the workspace, as they are all below the desired

waypoint. So we can send the robot to this configuration without worrying about colliding with anything in the workspace using the *safe move to position()*. Then, once at the waypoint, we know that the path between the waypoint and the desired end effector frame is clear, as long as the gripper is open. So we can use another *safe move to position()* to lower the arm to the block, at which point we can close the gripper and pick up the block. This method saves significant amounts of computing time and time spent moving the arm, as it reduces the path of the arm to two distinct motions. However, due to the nature of the waypoints and the design of the game, it doesn't sacrifice any safety or possible collisions with any blocks in the workspace.

0.1.4 Place block

Our approach to placing blocks is to construct a single stack of blocks centered in the center of the target platform. This done by our *robotPlaceBlock()* function which takes in the robot arm's team and the number of blocks already in the stack as input arguments. The robot's team determines whether the target y-value of the placement in the robot's frame would be -0.169 m (blue team) or +0.169 m (red team). The number of blocks already in the stack determines the value of the target z-value. For the first block in the stack, the target z-value is 0.230 m which res presents the height of the target platform (0.2 m) plus half the height of the block (0.025 m) and a clearance height (0.005 m). For each additional block in the stack, 0.05 m representing the height of that block is added to the target z-value. The target x-value for placing the blocks is 0.562 m for both teams. The center of the platform was chosen as the location to build the stack because if a block were knocked off the stack from a random direction, it would have the greatest chance of staying on the platform if it were already at the middle. For placing a block was always the end effector oriented as if it had rolled π radians about the robot's x-axis. We could keep this orientation consistent because the logic in *pickUpBlock()* always grabbed the block such that the white side of the block was already facing up or *rotateBlock()* would be able to rotate the block such that the white side would be facing up.

To place the block, the robot arm moves to a waypoint 0.075 m above (+ along the z-axis) the placement position. The arm's joint configuration for this waypoint is calculated using the neutral configuration seed. The purpose of this waypoint is to make sure the robot arm or block to not collide with any obstacles or the already-constructed stack. Our tests showed that including this waypoint accomplished avoiding collisions. After reaching the waypoint, the arm moves to its target pose, opens its gripper, and moves back up to the waypoint. The purpose of this last step is to avoid collisions with the arm and the placed block when the arm needs to move to grab another block. The configuration for the arm in each of these poses was calculated using our *solveIK()* method from a previous lab using seeds that were found beforehand which represented the arm in placing orientation at a height (z-value) of 0.230 m.

0.1.5 Rotate block

The *rotateBlock*() function rotates an already-stacked block $+\pi/2$ radians or $-\pi/2$ radians about the robot's y-axis. As inputs, this function takes in the robot's team, the number of blocks in the stack, and the transformation matrix of the block to the robot. The robot's team informs the function of the block's position along the y-axis, using the same values as the *robotPlaceBlock()* function. The number of blocks in the stack informs the robot the height of the top block it needs to rotate. This height is calculated in the same manner as the height in the *robotPlaceBlock()* function.

In practice, first determining which rotations (if any) were necessary for Side Bonus points, required knowledge of which world-frame direction the block's white side was facing after translation to the target platform. Thankfully, the implementations of *getPickUpTransformations()* and *robotPlaceBlock()* restricted these post-translation options to just 4 of the cardinal directions in the world frame. To determine which one, we recorded the EE's orientation relative to the block (via block2EE rotation matrix) utilized in the previous *pickUpBlock()* call, then the EE's orientation relative to the world (via world2EE rotation matrix) utilized by

the previous *robotPlaceBlock()* call, and combining them (via matrix inversion and multiplication) to yield the the desired block2world rotation matrix. Right multiplying by $[0,0,1]^T$ (i.e. white side orientation in block frame) gets the world frame coordinates, letting us case on the 4 possible values as follows:
Case +x: Rotate once -90 degrees about the y axis
Case -x: Rotate once 90 degrees about the y axis
Case +z: Don't rotate
Case -z: Rotate twice, 90 degrees about the y axis

The arm rotates the block in either direction by reaching a series of waypoints in a specific order. For example, to rotate the block $+\pi/2$ radians about the robot's y-axis, the robot first moves to a waypoint 0.05 m above the block at an orientation that represents a π roll about the x-axis and a $-\pi/4$ pitch about the y-axis and opens its gripper. It then moves down to the height of the midpoint of the block, closes its gripper, moves back up to the waypoint, and at the same spatial coordinate adjusts to a position representing a π roll about the x-axis and a 0 radian pitch about the y-axis. It this adjusts to an orientation representing a π roll about the x-axis and a $+\pi/4$ radian pitch about the y-axis. The purpose of these in-air orientation adjustments is to reduce the chance of a collision with the stack or platform during the block's rotation. The arm then moves down to the original height of the block at this same orientation, opens its gripper, and then moves back up to the waypoint to avoid block collisions.The method for rotating the block $-\pi/2$ radians is the same, except the ordering of the waypoints is reversed. The positions of the arm's joint at each of these waypoints are calculated using the *solveIK()* method and a dictionary of seeds that were found beforehand using the spatial coordinate of a first block in the stack and each of the three orientations for the waypoints.

0.2 Evaluation

Were all relevant results reported? Are the cases chosen sufficient to demonstrate advantages and limitations? In order to test and evaluate our code, we broke down our approach into 4 components: 1. unit testing 2. simulation 3. single pick and place and 4. full simulation.

0.2.1 Unit Tests

For unit tests, we broke down each function and performed print tests to evaluate if the returned output matched our expectations. Given the numerical nature of the outputs of each function, we weren't able to initially check if each index of the returned output exactly matched what we would expect. However, we were able to look for specific indications in the desired transformation matrices to check their validity.

For example, in our *getTags* function, we first printed out a single instance of the output from $get\_detections()$ and ran our *getTags* on that data to avoid confusion with constantly changing block configurations. When we printed the output of the *getTags* function, the orientation of tag in the robot frame was difficult to verify. However, from the base of the robot, we had a rough estimate of the x, y, and z coordinates of the tag in the robot frame from the given dimensions in the final project instructions. Depending on the team, the center of the top platform that held the static blocks had a position vector of $[0.562, 1.147, 0.200]^T$ in the world coordinate frame. Given that the robot base was 0.978 m in the world y-axis, the static bottom of the blocks should be around $[0.562, 0.169, 0.200]^T$. Note that the x and y coordinate signs may be negated based on the team. Because the transformation was the top tag of the block, we needed to add an additional 0.05 m to the z coordinate. We verified that the transformation matrices from *getTags* had a position vector around the expected value. This method of evaluation allowed us to catch some initial errors in our dictionary of T_{tagi}^{block} transformations.

As another example, we did something similar with the position vectors from our *getBlocks* function with the exception that we expected the z coordinate to be around 0.225 m because we the center of the block

is 0.025 m below the tag. Another method of verifying our output was checking that the value in the third row, third column of the rotation matrix was around 0, 1, or -1 because we knew that the block's z-axis had to either be in the same direction as the robot's z-axis, opposite direction, or along the plane perpendicular to the robot's z-axis.

Smaller functions also allowed for more direct testing. For example, our *getOrderedBlocks* function takes could be tested by doing a manual check for the first ordered list based on an estimate of which blocks are closer to the robot base. Then, we could manipulate the order of the transformations going into the function and verify that the output remained the same ordered list.

As our functions included more complicated logic dependent on commands sending the robot arm to a specific configuration, we moved on to simulation tests.

0.2.2 Simulation

After doing our unit tests on static data, we ran our functions in simulation. We planned to compartmentalize our testing to build on top of each other. We initially planned to use RRT, but in simulation, we realized that the random sampling of the configuration space without significant obstacles would often command the robot to an undesirable configuration, sometimes in the opposite direction of the static block platform. Therefore, we removed RRT planning and focused on using waypoints. Therefore, we first planned to simulate the robot going to a waypoint and then to the closest block on the static platform. Next, we wanted to simulate getting to the block and picking up the block. Then, we wanted to simulate a single block pick and place with a follow up simulation to test the full three minute run time on all four static blocks.

One of the limitations of our simulation evaluations was that none of our virtual environments had a real time factor greater than 0.1 which would result in over 20 minute runs in order to find individual bugs. Therefore, most of our meaning simulation testing occurred during our in person lab testing time.

During our first in person lab testing, we ran into errors we weren't getting in our personal virtual environments that tied to the *calculateFK* function in our lib folder. We decided to stick with the lib folder we initially chose and modify as necessary to fix the errors. However, after running into additional errors in our second lab section, we decided to switch lib folders. At this point, the code was relying on a specific function that was only in the original lib folder, so we formed a combined lib folder.

From simulating the movement of the arm to the waypoint and then to the closest block on the static platform , we found that our initial waypoint above the static platform was causing the end effector to rotate unnecessarily along the world's z-axis as it was approaching the static block. Therefore, we moved the waypoint lower in the z-direction and added waypoints right above each static block. As our initial unit tests were run on a single test environment with two top facing "tag3" and two top facing "tag4" blocks, simulations that included a upward facing "tag5" led the end effector to be slightly off from the center of the block when grasping as shown in this demonstration. This allowed us to find an error in the T_{tag5}^{block} transformation.

0.2.3 Single Pick and Place

After our simulation continually picked up and dropped the first static block, we tested this on the robot during our second in person testing session. Our main objectives for testing were:

1. Eliminate potential bugs that would occur on the physical system but not during simulation
2. Determine the actual amount that the first block would bounce when dropped from the first placing waypoint
3. Get a sense for the speed of the maneuvers

After resolving bugs in simulation, the physical system did not result in significant bugs for the single pick and place challenge. However, when we dropped the first static block from a significant height, we noticed that although the orientation would remain within a ± 15 deg angle and the position was within ± 1 cm from the center of the final platform. This motivated us to create a target drop position closer to the height of the final platform, so we used the inverse kinematics function to find the configuration such that the block would not be touching, but right above the final platform.

From testing on the physical system, we noticed that the movement of the arm when commanded to a specific joint configuration was reasonably smooth such that there were no sporadic jolts and the speed was comparable to what we had seen in other teams' demonstrations. However, there seemed to be significant pauses while the algorithm was calculating the next target configuration.

0.2.4 Full Testing

After testing our single place and pick, we moved onto our full game simulation and physical testing. We planned to split the testing into two phases: 1. testing by pure placement without rotating the goal face up and 2. testing with goal top target so that we could focus on the placement functionality and add additional complications of rotation when we were confident in our placement ability.

For the first phase, we were looking for the ability to drop the blocks without significant sliding such that the tower was relatively straight even after the four static blocks had been stacked. We also looked wanted to get a baselines for how long it took just to stack all the static blocks in their original configuration and check that our algorithm correctly determined the configurations necessary to move between the final and static block platforms.

The first phase worked smoothly in simulation, and during testing, we found that it took less than 3 minutes which encouraged us to include a rotation functionality to get the side bonuses. When testing on the physical system, we found that the significant number of waypoints above the static platform, right above each static block, above the final platform, right above the previously stacked block, and above the final platform significantly increased the amount of time it took to stack each block, so we removed some of these intermediate waypoints.

For the second phase, we were looking for the ability to correctly determine how many rotations were necessary in order for the goal face to point in the positive world z-axis. Initially, the rotations were occurring in the opposite direction such that the goal face was getting further away from pointing in the positive world z-axis, but after some further testing, we were able to accurately determine and execute the desired rotations necessary to get the desired result.

$$\begin{bmatrix} 0 & -1 & 0 & -2 & 0 & 1.57 & 0 \\ -1.9665 & -1.3502 & 2.2974 & -2.1726 & 1.6535 & 1.5157 & 2.5838 \\ -1.2 & 1.57 & 1.57 & -2.07 & -1.57 & 1.57 & 0.7 \end{bmatrix}$$

In simulation, we were able to successfully stack all four blocks with their white sides facing upwards with an average time of 4 minutes 30 seconds. This is above the competition time of 3 minutes, however we attributed much of this time to delays in opening and closing the gripper that do not occur on the physical robot. An image of this successful stacking can be seen in Figure 1 in the Appendix. It should be noted that the stack is not exactly completely straight which is due to the simulation gripper not always gripping the block symmetrically.

0.3 Analysis

In our analysis, there were two factors we could really attempt to measure through all of our testing. First was obviously success - does the function we are testing work successfully? Are we successfully able to

go to the block and close the gripper around the block? Can we successfully rotate the block 90 degrees in either direction? These factors are obviously the most important, but fairly easy to measure either yes or no.

More importantly, the second factor we wanted to measure was speed. How long does it take the robot to accomplish one function? How long does one entire pick and place sequence to occur? How long does it take to stack all 4 static blocks in a white-side up orientation? These are much more data driven and based on the results we can optimize our code. However, it was often difficult for us to test timing issues. No one in our group had a very good Virtual Machine on their computers - thus, when testing in simulation, it was hard to get a great grasp of exactly how long functions were taking. We were operating on real-time factors anywhere from 0.03-0.1, and it would change over the course of a single simulation. Additionally, the simulation time and real time values on the bottom did not always seem to be correct and line up with the real-time factors we were reading. Therefore, we had to do most of our timing tests in lab, either in simulation on the computers which a much high real-time factor or on the actual robot itself.

To begin, we tested our ability to find the block orientations and calculate the desired end effector frames necessary to pick up these blocks. These were relatively simple mathematical computations involving transformation matrices and linear algebra, as explained above. So in simulation, we saw that this process took a matter of milliseconds - a negligible amount. Additionally, this process only occurs once at the beginning of our code. After seeing how little time it took in simulation and in physical testing, as well as seeing it's success in testing, we were confident with this section of the code.

Next, and more importantly, we tested the *pickUpBlock()* function. As stated above, we initially had this function built with RRT. During our initial tests for the RRT method, we saw how long the RRT took and how randomly the robot moved across the workspace. To move between the neutral configuration and a block location, due to the randomness it could take anywhere between 10 seconds and over a minute as seen in our testing. Obviously this is infeasible, given the 3 minute time constraint for the project. Therefore, we scrapped the RRT and moved to the waypoint method.

Initially, we had two additional waypoints in our path, which were 0.5m above the center of the static block platform and the goal platform. The idea was that if every time we move from one platform to the other we go up to these waypoints, the robot will never collide with any blocks or towers we may build. This was proven to be true - we never collided with any blocks or towers during our testing. However, with these very conservative waypoints time became a big issue. We ran a test where we stacked 4 blocks in a tower, regardless of orientation, using these extreme waypoints. This process took 2 minutes and 34 seconds - and this was before we implemented any block rotations, which would only add to the time. Moving the arm all the way up, over, and then all the way back down was very costly time wise. Therefore, we decided to try to make the path between the two platforms a lot more linear. We still needed waypoints - we tested without waypoints, and we would either knock down the tower or the gripper hands would collide with the block when moving towards/away from it. But we instead used waypoints just above the goal positions for the robot. This helped make the path of the arm a lot more linear, and brought our trial time to 1 minute 23 seconds (again, this was without any rotations or re-orientations of the block).

Next, we wanted to test the time it takes to rotate a block 90 degrees. Our initial rotation code has a series of waypoints over the course of the rotation period to ensure the block rotated safely. In testing, we found that this rotated a block 90 degrees in 44 seconds. We found that this was way too slow for our efforts. We estimated that a given round would require at least 4 total block rotations, in addition to stacking each block to achieve our goal. With the stacking time listed above and only 4 rotations, this would already take 4 minutes and 19 seconds, well beyond the time limits for the competition. Therefore, we needed to find a way to speed up the rotation process.

By the time of the competition, we lowered the time to rotate to 27 seconds. We did this by decreasing the number of waypoints in the rotate code to only the critically essential points to rotate the block. During the competition, our first trial stacked all of the white blocks successfully in 3 minutes and 2 seconds - just longer than the given time period. In the second trial, we failed stacking the second block, when it appeared

that the block had collided with the tower when attempting to rotate the block. We hadn't seen this at all during any testing of the rotating code, even with the limited amount of waypoints. We are attributing this most likely to the placement of the first block in the tower being slightly off due to new conditions in Wu and Chen compared to B2. If the location of the first block is slightly different, it may be in the path of the rotation, which caused the failure.

After seeing our results in the competition, there are a few things we can do to attempt to improve our performance. First off obviously is speed - how can we make our process faster. We believe that we had limited the number of waypoints down to the absolute minimum we could. If we got rid of any more waypoints or changed the location of the waypoints, we would run the risk of colliding with the blocks or towers when moving. Therefore, we needed to speed up how we move between two locations. In our code, we had chosen to use *safe move to position()* for simplicity - it only requires a goal configuration and then moves directly to that configuration. However, if we had used *safe set joint positions velocities()*, we think that we could have had the arms move much faster. By setting the velocities, we could have the robot arm move at a much faster rate than it does with *safe move to position()*, which would obviously decrease the amount of time our simulation took. Smaller movements of the arm, such as from the waypoint directly above the block to the block itself, may not benefit from setting the velocities. However, larger movements, such as movement between the two platforms, could have been sped up by setting the joint velocities.

Another idea is to alter the way we placed the blocks. We had always wanted to pick up the blocks in a top down orientation, as opposed to grabbing from the side. In the sideways orientation, we are very likely to collide with other blocks on the table or even the platform itself, given that the arm diameter is wider than the size of the block. However, we limited ourselves to place the block top down as well, after which we would have to rotate the block. We did this because of the same concerns of colliding with other blocks or the platform. However, we may have actually be able to circumvent this problem. By carefully utilizing waypoints and dropping blocks slightly above the desired locations, we could have avoided colliding with anything. This would greatly save the time it takes to place a block. For example - a block with the white side on the side (tag 1, 2, 3, or 4 up) would need to move the block from the static platform to the goal platform, and then need to rotate the block once. The movement of the block would require at least 4 waypoints, and then the rotation would require another 6, meaning the arm would have to move 10 distinct times to place and orient this block correctly. However, if we instead placed the block on its side instead of top down, we could lower the number of waypoints to 5 total, halving the number of times the arm has to move. Additionally, for a block that is tag5 up, instead of rotating the block twice, we could first place it on its side and then only have to rotate it once, again decreasing the number of waypoints and time it takes the block to move. By placing the block on its side instead of top down, we can greatly reduce the number of waypoints we use and speed up the process a lot, which would help our performance greatly.

0.4 Lessons Learned

0.4.1 Strategy Implementation

As mentioned previously, we initially included RRT trajectory planning but then replaced it with waypoints. This led to us removing several hours of work and testing from the code base that we could have avoided from understanding that the lack of significant obstacles did not require the use of a planner. This time could have been much better spent finding more efficient waypoint configurations and standardizing the configurations used to rotate the block 90deg when trying to get the goal face bonus.

One aspect of our strategy that we changed but likely resulted in negative consequences was only grasping the blocks from the top. We initially thought that grasping the blocks from the sides could allow for direct stacking of the bonus face pointing up on the final platform. Due to our concerns that we were not running any collision detection, we decided to pivot to grasping the blocks only from the top to limit the

probability that we would collide with the platform or other blocks when picking up a block. However, always grabbing and placing the block from the top made placing the block with the side bonus take a significant amount of time.

From the initial criteria and scoring system, stacking the dynamic blocks at the end seemed to provide the highest scoring potential because the higher the altitude of the dynamic blocks, the higher the score for that particular block. After the rule change that the dynamic blocks would not be included in the $get\_detections()$ output, we maintained the same strategy of first stacking the static blocks without significant conversation about a potential change in our strategy in response to the change in the competition. After looking at other team's performance on the day of the competition, stacking the dynamic blocks first seemed to provide significant benefits. Not only was the bonus face already facing up, but first stacking the dynamic blocks would also likely reorient the leftover dynamic blocks that were still on the rotating table, decreasing the probability that other teams could also stack dynamic blocks.

From another team's presentation, we also learned that we had made the assumption that the blocks would be spaced far enough apart such that the gripper going for a block would not collide with another neighboring block. Although this did not limit our scoring potential in this competition, this could have led to major limitations in our design if the beginning static block configuration had the blocks clustered together.

0.4.2 Project Logistics

One of the challenges of the project was code organization. There were four of us on the team, each with our own machines and assigned tasks. This meant we had to establish a shared code repository and conventions for sharing code to avoid issues arising from inconsistency between team member's code. Due to some team members' unfamiliarity with git, we decided to use a shared google drive as our code repository. We implemented a naming convention to help with version control and assigned modular tasks to avoid redundant work being done and encourage specialization/expertise. This scheme worked well enough for our purposes, as we were able to accomplish our functionality goals, however we definitely learned a lesson on organization.

The main difficulty was maintaining version solidarity between each team member's code. Each team member was concurrently working on their assigned function. To run general/compilation tests on the code meant each person's latest code to be manually assembled into a single final.py file. This was doable, especially since our team had excellent communication over text, but it was definitely an inconvenience and led to errors a few times. This problem would have been made easier with git's push/pull functionality. If we were to do this project again, we would probably just learn and use git.

A secondary and perhaps unavoidable difficulty was the reliance of downstream functions on upstream functions. Each person's assigned function was dependent on the previous function's functionality. For example, we could not verify if the place block function could construct a stack of blocks until we could verify that the pick up blocks function we could reliably pick up blocks. This difficulty mainly presented itself if the testing phase of each function's construction, yet was mainly mitigated by us defining very specifically what each function's inputs, outputs, and purpose would be at the beginning of the project.

As always, we would have been served well by beginning the project earlier than we did. We began planning work on the project approximately two weeks before the competition date, but didn't start physical tests on the robot until the week of the competition.

0.5 Appendix

0.5.1 Video Links

Competition round one: perfect stack of four white-up blocks, in 3:02 time: `https://youtu.be/bhfcxOF-z2I`
Furthest block hardcoded, high drop which bounced: `https://youtu.be/gIQfSv7XI9M`
Rotation from white down to white side: `https://youtu.be/aNJdSvsmX98`
Simulation gripping block center, getBlocks() bug fixed: `https://youtu.be/LKdkVjIPhyY`
Physical test grabbing error: `https://youtu.be/cfPvo6HFG9s`
Simulation Placing Block: `https://youtu.be/M3CWIMF9ufc`
Simulation Picking Up Block: `https://youtu.be/ed5a7Gw92P0`
Physical Stack 4 Blocks Without Orientation: `https://youtu.be/HJBDek4swnM`

0.5.2 Figures

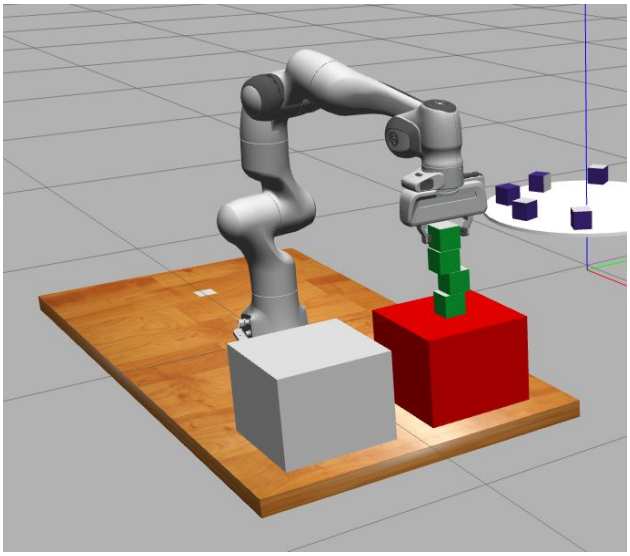

Figure 1: Full stack of properly oriented static blocks in the sim