# MEAM 5200 Final Report

Ariana Kyimpopkin, Irene Grace Karot Polson, Ojas Ajitkumar Mandlecha, Sarah Ho

December 2022

## 1 Strategy

At a high level, we planned to separate the targeting of static and dynamic blocks into two subfunctions, `staticBlocksCenter(team)` and `dynamicBlocksCenter(team)`, allowing for modular testing and rapid adjustment of our strategy.

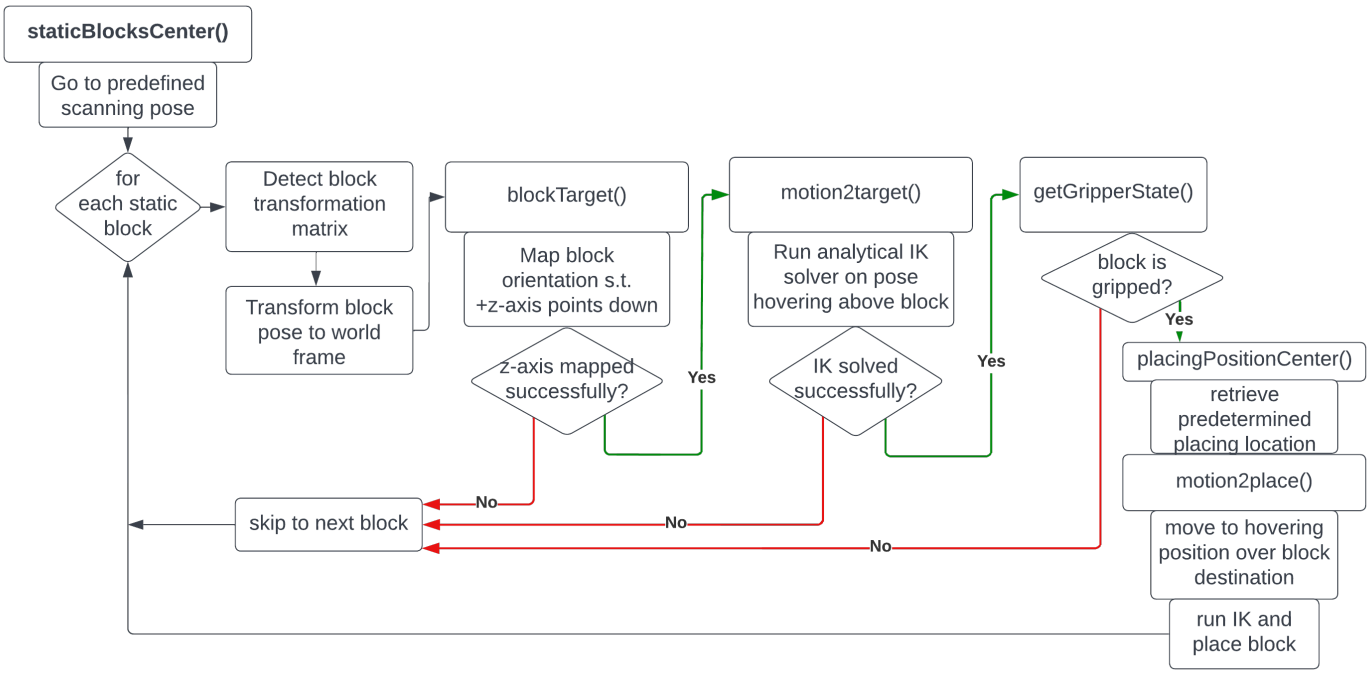
Our minimum viable performance was code which could successfully stack static blocks in hardware, which we achieved during lab time and, to some extent, during the competition (see the Results section).

After developing code to achieve this, we wrote functions for dynamic block stacking as we continued to test code for handling both types of blocks.

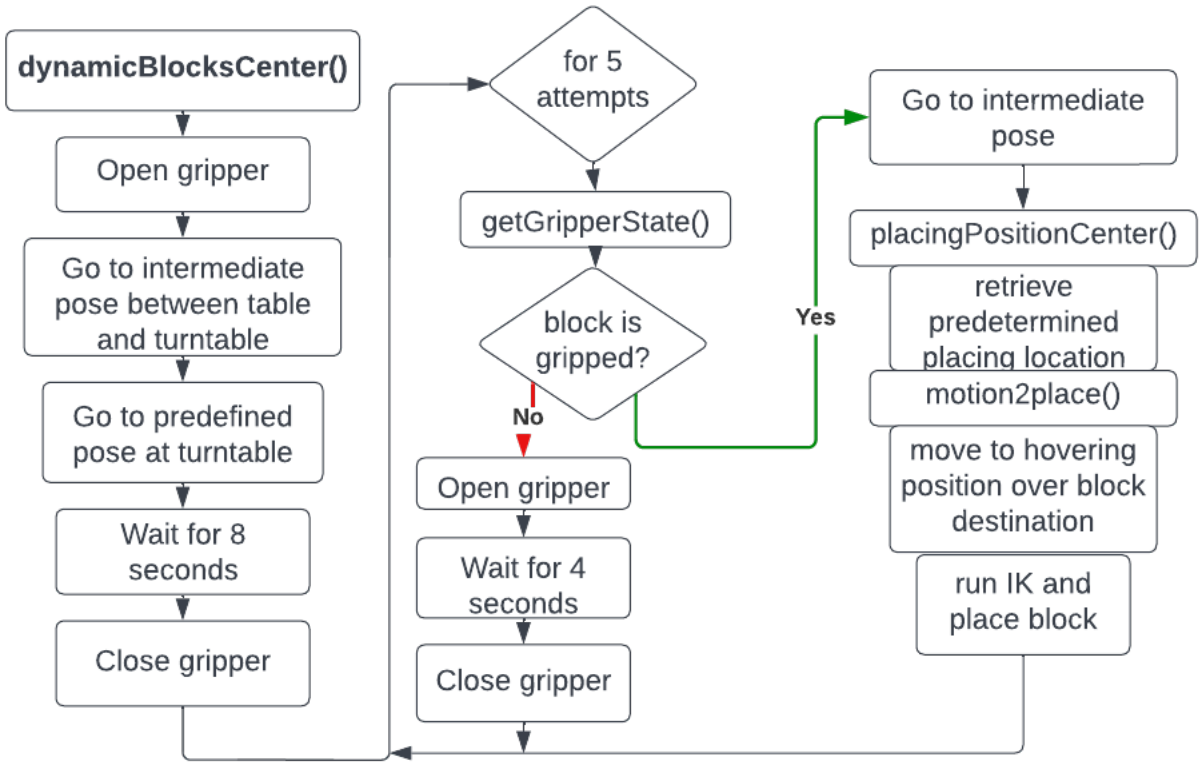
Though we didn't end up using it in competition, we also developed a subfunction called `dynamicBlocksSide()`, which retrieves a dynamic block at the beginning of the match, places it safely to the side on the table, then adds it to the top of the stack after stacking all static blocks to maximize points. This was considering if the opponent team tried to sabotage the dynamics blocks while we were busy stacking static blocks.

## 2 Method

### 2.1 Overall Code Flow for Static Blocks



## 2.2 Overall Code Flow for Dynamic Blocks




## 2.3 Scanning Positions

In order for the robot to receive the position of an arbitrary block via its AprilTag, that block must be in the camera's field of view. Therefore, we hard-coded an initial configuration that allows the camera to see the whole of the goal and starting platforms as well as the static blocks. When the robot is in a scanning position, therefore, it reports the H-matrix of all static or dynamic blocks regardless of their position/rotation.
In simulation, using the r-viz utility, we determined these positions to be:

$$\vec{q}_{scan,stat} = ()$$

The simulated FOV of each scanning configuration is shown below. Note that all of the platform is visible in each case.

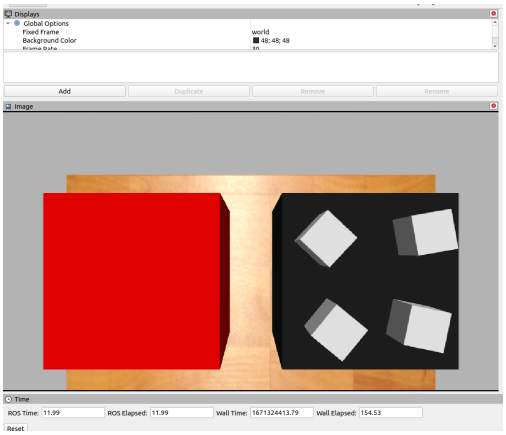


Figure 1: Static scanning FOV

## 2.4 FK Transforming Block Matrix to World

The AprilTag `ObjectDetector()` class reports the transformation matrices of all visible blocks relative to the camera frame. We will refer to this matrix as $H_b^c$ going forward. In order to make use of our IK code – and understand the real locations of the blocks – we must transform this matrix into world-frame coordinates.

This requires multiplying the AprilTag output with the transformation from camera to effector frame (which is fixed by the camera and robot interface) $H_c^7$ and then with the transformation from the effector the world frame $H_7^0$ (which varies depending on the configuration of the robot).

$$H_{block}^0 = H_7^0 * H_c^7 * H_b^c$$

In our code, we obtained $H_c^7$ from the `H_ee_camera = detector.get_H_ee_camera()` method, and $H_7^0$ from the `calculateFK.py` class generated for Lab 1.

See the diagram below for a summary of this process.

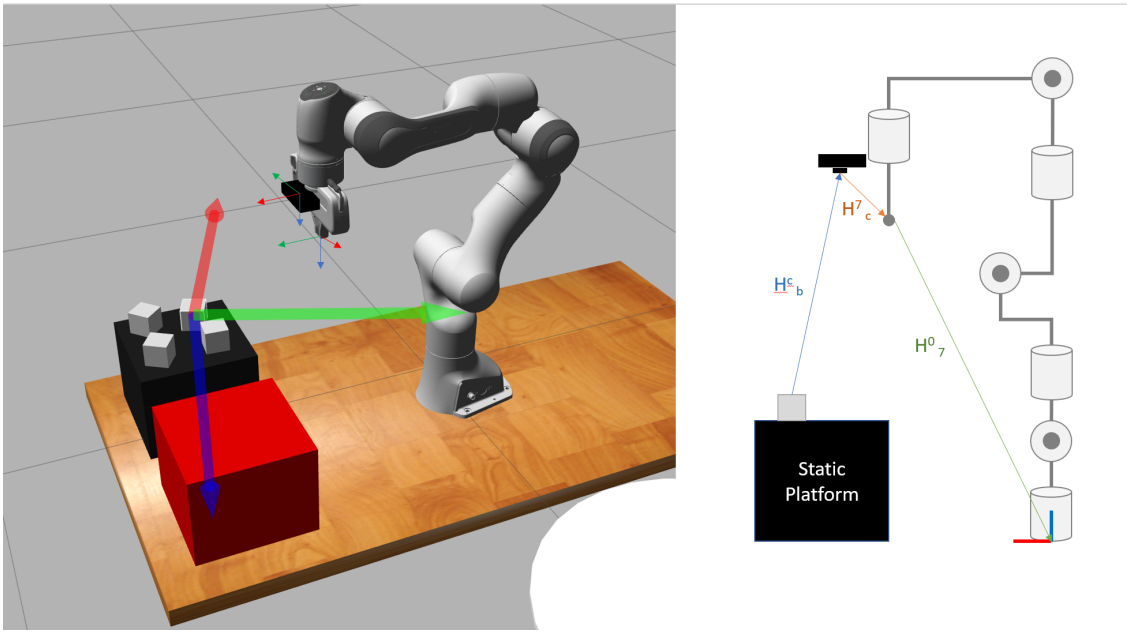



Figure 2: Frames involved in robot FK in simulation (left) and diagram of transformations (right)

## 2.5 IK Methods

Once we had the location of all blocks in the world frame, it was necessary to perform inverse kinematics in order to find a configuration which would align the gripper frame with the block frame to facilitate grasping.

In order to be useful in the case of real robot control, our IK solution needed to include a number of functionalities beyond those required by Lab 2:

1. Ability to constrain the solutions to those inside the joint limits of the PANDA arm.
2. Single solutions anywhere in the workspace OR a means to select the most optimal solution from a list.
3. 7 DOF IK capability, as compared with the 6DOF Lab 2 IK that held joint 5 fixed.
4. Quick runtimes OR the ability to run during robot motion.

Initially, we considered – and built a simple version of – a gradient descent IK solver using the force and update law below. While this approach improved runtime greatly, and was able to handle 7 DOF IK, we were not able to determine an efficient way to constrain the solution set to legal joint-angle configurations only. In addition, we determined that joint 5 motion was not crucial to the function of the robot given that we were primarily approaching from the world z, placing joint 7 inline with joint 5.

$$\nabla f(q) = || \vec{o}_n^0(q) - \vec{o} \;\; ||^2 \implies \Delta q = -\alpha \, J_v^T ( \vec{o}_n^0(q) - \vec{o} \,)$$

As a result, we chose to use a version of the Lab 2 IK code with additional logic added to filter out any solutions which violated the joint limits.

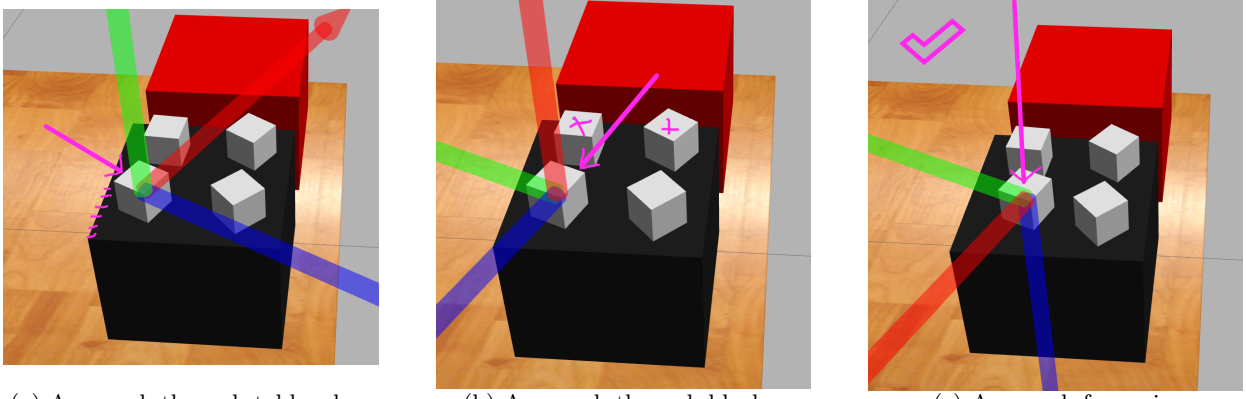
```
#JOINT LIMIT
for i in range(q.shape[0]):
    if q[i,0] > 2.89730 or q[i,0] < -2.89730 or q[i,1] > 1.76280 or q[i,1] < -1.76280 or q[i,2] > 2.89730 or
        #assign to NaNs that will be filtered out later
        q[i] = np.nan

#remove rows with NaN elements
a =np.argwhere(np.isnan(q))
idx = a[:,0]
q = np.delete(q, idx, axis =0)
```

Figure 3: Joint limit filter code excerpt

Eventually, we were able to enforce the creation of only 'safe' top-down gripping configurations by ensuring that the z component of the block frame was always antiparallel to the z axis of the world. This allowed us to select any solution when a position had multiple configuration-space solutions.

## 2.6 Block Frame Rotation

Whatever the specifics of the IK solver, the process seeks to produce a configuration that aligns the axes of the gripper frame with the block frame being solved for. On the Panda arm, the effector z points normal to the gripper. Therefore, unless the z axis of the gripper points down into the table, the robot may approach from a non-safe direction, resulting in collision with the table or other blocks as shown below.



(a) Approach through table edge (b) Approach through blocks (c) Approach from air

Figure 4: Gripper approach and axes relationships

As such, we included a subfunction `blocktarget(T0b)` to transform the block H matrices into a configuration where z points down before calculating the IK configuration.

1. Approach 1: As a basic test, we first tried setting the rotation component of the block transformation matrix in the world frame $H_b = \begin{bmatrix} \vec{R} & \vec{d} \\ 0 & 1 \end{bmatrix}$ to $\vec{R} = \begin{smallmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{smallmatrix}$. This verified that having a z-down configuration consistently resulted in safe approaches. However, because orientation did not always match that of the block, it was not robust for block grasping.

| Starting Orientation | Transformation Required | Rotation Matrix |
|---|---|---|
| X axis up | -90 about y | $R_y(\frac{-\pi}{2})$ |
| X axis down | +90 about y | $R_y(\frac{\pi}{2})$ |
| Y axis up | +90 about x | $R_x(\frac{\pi}{2})$ |
| Y axis down | -90 about x | $R_x(\frac{-\pi}{2})$ |
| Z axis up | 180 about x | $R_x(\pi)$ |
| Z axis down | no transformation needed | $I_{3x3}$ |

Table 1: Block Frame Transformation

2. Approach 2: Next, we applied intermediate-frame rotations about y or x to the transformation matrices based on their staring configuration so that the resultant $H_b$ retained its in-plane orientation.

$$R_y = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix} \quad \text{and} \quad R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) \\ 0 & sin(\theta) & cos(\theta) \end{bmatrix}.$$

Using an if-else structure, we identified the axis which was parallel to world z (including a tolerance value epsilon for noise handling), then applied the appropriate rotation by post multiplication.

```
def blockTarget(T0b):
    #Code to transform the axis of block
    R = T0b[0:3,0:3]
    print('R:', R)

    last_row_plus = np.absolute(R[2,:] - 1)
    last_row_minus = np.absolute(R[2,:] + 1)
    epsilon = 0.01

    if last_row_plus[0] < epsilon:

        print("xup")
        theta = -pi/2
        R_y_90 = np.array([[cos(theta), 0, sin(theta)],[0, 1, 0],[-sin(theta), 0, cos(theta)]])
        R_new = R@R_y_90
```

Figure 5: blockTarget code excerpt

The rotations required for each orientation are shown in the table below.

## 2.7 Gripping

Within `motion2Target()`, after IK-solving for a reachable position above the target block, we grip the block with the command `arm.exec_gripper_cmd()` with distance = 0.025m and force = 30N, values experimentally determined during lab sessions.

After commanding the gripper to close, we also check that a block has actually been grasped using `getGripperState()`. This method uses `arm.get_gripper_state()` to check that the true position of the gripper indicates that a wooden block is being held securely on its sides.

Given the mounting angle of the camera, it cannot be used to verify the position of the block in the gripper. However, we incorporated code to check that a block has been stacked before incrementing the goal position to avoid 'skipping over' failed blocks.

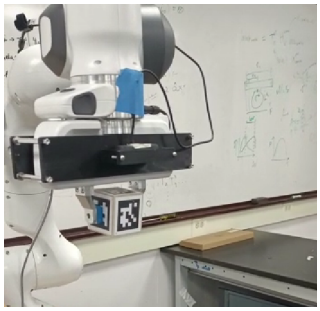


Figure 6: Gripped block

## 2.8 Path Planning

At this point, our code was able to calculate configurations that corresponded
to a safe scan, block grasping, and block stacking. We had also selected a center position on the goal platform, which would be validated in lab. Next, we had to consider how to move between these configurations while avoiding obstacles. It was desirable to have a quick calculation time.
The two options were considered were:

- Potential field -based path planning that used lab 4 code as a jumping-off point. This had the advantage of built-in obstacle handling, but would have a) resulted in longer calculation times and b) been less repeatable due to the effects of random walks.
- Going directly to configurations using `arm.safe_move_to_position()`. This does not take into account obstacle avoidance, but is faster and repeatable.

Because there are few obstacles in the workspace (namely any existing stack of blocks, the tables, and the turntable), we command directly from one pose to the next without artificial potential fields or similar path planning methods in order to maximize speed of performance within a tightly timed match. Since we had chosen the more time-consuming geometric method for IK, we felt it was important to balance this with a faster 'path-planning' methodology. For these two reasons, we selected the second option.

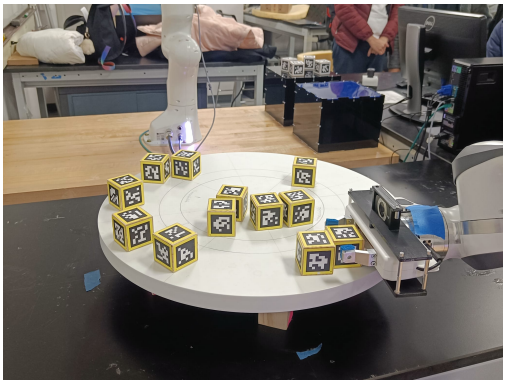
To address the obstacle downside of `arm.safe_move_to_position()` approach, we added an intermediate "hovering" waypoint instead of going directly pick or place a block to our `motion2target()` and `motion2place()` subfunctions. This prevents the end-effector from knocking into the edges of blocks when approaching from an undesirable angle. The "hovering" position is clear of any obstacles, and the end-effector needs only to move a short distance downward to pick or place, which it proceeds to do immediately after reaching this position.

Another feature used is a global variable i, that stands for the position on the stack where the picked up blocks need to be placed. This variable only gets incremented when the block is placed making sure that we do not accidently try to make placement after having dropped a block mid-path or after failing to pick up. However, as will be discussed under analysis, this did not allow for reaction to dynamic obstacles.

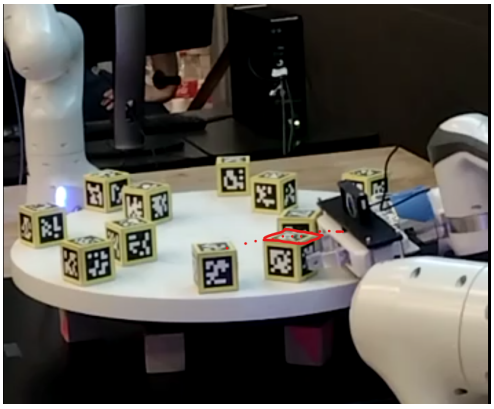
## 2.9 Dynamic Blocks

The routine `dynamicBlocksCenter()` moves the arm to a static position such that the turntable's rotational motion funnels the blocks into the gripper, which periodically (using the `time.sleep()` command) opens and closes the gripper until a check of its state indicates that it has closed around a block, then places it in the next location stored by `placingPositionCenter()`. With twelve dynamic cubes in play and distributed more or less evenly on the turntable, we counted on the fact that we were likely to encounter a block within a few seconds of turntable rotation.

We chose this strategy because moving to a hardcoded position at the turntable eliminates risk of the camera attachment knocking into the turntable or table while trying to match the variable orientation of a target block. This simplified the hardware testing we would've had to do to compare the camera attachment's size and shape in simulation versus real life. However, it risks not being at the correct orientation to actually grab any blocks, as we observed in the competition and discuss later.



(a) Successful block orientation



(b) Unsuccessful block orientation

Figure 7: Block Orientation during dynamic grasping

### 2.10 Error Handling

Try and Except statement is used to handle these errors within our code in Python. As our code had multiple modules, we had to use the above method to deal with errors in each module to be able to execute the next steps of our strategy so that our code does not get stuck during the competition and continues to execute the next step. In the code, we had hard coded the position values for stacking. But sometimes, like fail to grip or error, the block wont get picked but our variable 'i' which is used to define the block number use to get updated. So to avoid this we made 'i' a global variable in our code and updated only when the block has been gripped.

## 3 Evaluation

### 3.1 Simulation

We used simulation and hardware testing successively to develop new code over the course of three lab sessions. Once a milestone behavior – such as static block stacking – was demonstrable in simulation, we would try to duplicate it in a lab environment to understand the differences between the simulated and real environment as described in the next section.

### 3.2 Goals of Hardware Testing

We expected a number of differences between the simulation used to develop our code and the actual environment of the competition. Our testing focused on the hardware/simulation differences below, and ensuring we had the relevant information to design for the hardware environment.

1. Camera position and orientation on robot may not match simulation.
   Desired information: Real camera-to-effector transformation matrix $H_c^7$.

2. Real camera's reporting of the block orientation and position contains noise.
   Desired information: value of orientation tolerance `epsilon` parameter that consistently identifies the orientation of a block before transformation.

3. Weight and friction of real blocks differ from those in simulation. A gripper force value as part of the `exec.gripper_cmd(pos, force)` that identifies when a block as successfully grasped in simulation may not function in hardware.
   Desired information: value of the `force` parameter that is tuned to the real environment.

4. Our strategy involved hardcoding a base position of the block stack on the goal platform for the blue and red team cases.
   Desired information: Are these values appropriate to the real environment?

5. We also hardcoded 'scanning' positions that created full visibility of the static blocks.
   Desired information: Can the robot see the whole of the static platform? Are all blocks successfully scanned even in extreme cases?

6. Stacking blocks is a balance between avoiding collision (if the offset from the previous block is set too low) and avoiding block bouncing off the top of the stack, possibly collapsing it (if the offset is set too high). We needed to determine in hardware what the idea value of the offset is.
   Desired information: block center offset value

7. Our strategy for grasping dynamic blocks involved placing the gripper at a predetermined radius, then repeatedly closing the gripper until `getGripperState()` reported that a block had sucessfully been grabbed.
   Desired information: dynamic block position and grabbing time delay

8. Verify all robot (arm.method) commands used are compatible with the hardware setup.

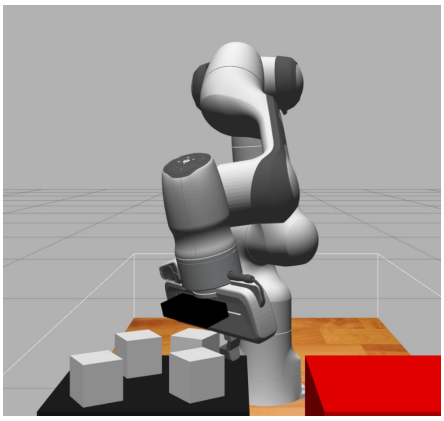
## 3.3 Hardware Experiments and Results

### 3.3.1 Lab Session 1 Test Cases and Results

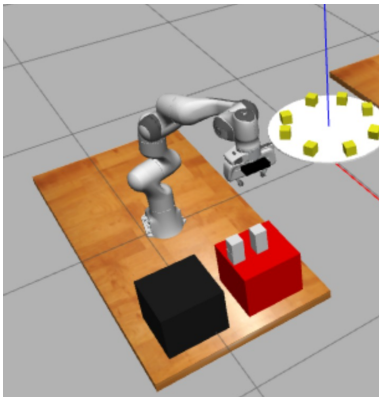
In our first lab session, our code, which worked inconsistently in simulation to pick up static blocks, did not succeed convincingly enough to run on hardware. We could transform the state of blocks to world coordinates, and perform IK to get a configuration which should have placed the effector on the block. However, our IK solutions often approached from an inconvenient direction, appearing to be misaligned by about 20-50mm from block center.

Discussing with a TA, we found that the problem was not primarily with our IK solver, as we'd thought, but the root cause was that the randomized block orientations did not have the z-axis pointing down, so the IK solver couldn't find reachable safe configurations to align the end-effector orientation with the blocks. Based on this feedback, we added the `blockTarget()` function to map the z-axis downward.

We tested this function in simulation on all possible block orientations (y up, y down, x up, x down, z up, and z down), and determined that it could consistently pick up blocks of all orientations. This allowed us to implement the stacking behavior below.



(a) Before lab session 1



(b) After lab session 1

Figure 8: Lab session 1 Evaluation progress

### 3.3.2 Lab Session 2 Test Cases and Results

In the next lab session, we had prepared four-static-block stacking code that was successful in simulation with multiple randomized block starting configurations. Our goal for this lab session was to verify points 1-6 in hardware.

Now that we were remapping the block orientations to force an aerial approach, our IK and path planning performed without incident, verifying the strategy for static blocks that had previously been demonstrated to work in simulation (Fig 8b).

However, the `open_gripper()` and `close_gripper()` commands did not work on hardware; we switched to using the `exec_gripper_cmd(pos,force)` command instead to be compatible with the hardware.

After this error was addressed, we were able to verify that our transformation matrices for the real camera were correct, verify the static scanning position, select values for the `cmd` and `pos` parameters in the gripper code, and verify that the $q_{scan,static}$ configuration provided full visibility. To address the issue of noise in the AprilTag reading, we added a tolerance value $\epsilon$: if an axis was within epsilon of parallelity with the world z axis, it was considered to be pointing up.

Finally, we tested the hardcoded positions for the blocks in each level of the tower to verify they were accurate to the hardware setup. We determined that a 0.025m offset was sufficient to prevent collision with the table, and adjusted our hardcoded positions accordingly. The full table is shown below:

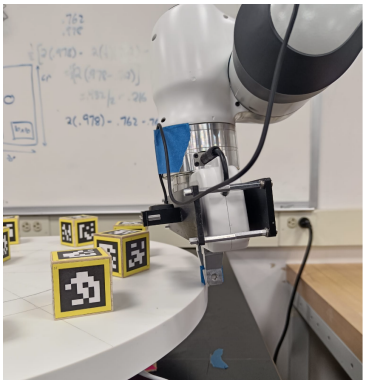
| Block Different Positions | | | | | |
|---|---|---|---|---|---|
| Sr No. | Type | X | Y | Z | Rotation Matrix |
| 1 | All static on platform | 0.5245 | 0.1685 | 0.225 | [1,0,0,0,-1,0,0,0,-1] |
| | | 0.5245 | 0.0935 | 0.225 | [1,0,0,0,-1,0,0,0,-1] |
| | | 0.5995 | 0.1685 | 0.225 | [1,0,0,0,-1,0,0,0,-1] |
| | | 0.5995 | 0.0935 | 0.225 | [1,0,0,0,-1,0,0,0,-1] |
| | | | | | |
| 2 | Stack 2 Static blocks | 0.562 | 0.181 | 0.225 | [1,0,0,0,-1,0,0,0,-1] |
| | | 0.562 | 0.081 | 0.225 | [1,0,0,0,-1,0,0,0,-1] |
| | | 0.562 | 0.181 | 0.275 | [1,0,0,0,-1,0,0,0,-1] |
| | | 0.562 | 0.081 | 0.275 | [1,0,0,0,-1,0,0,0,-1] |

Figure 9: dynamic block in gripper

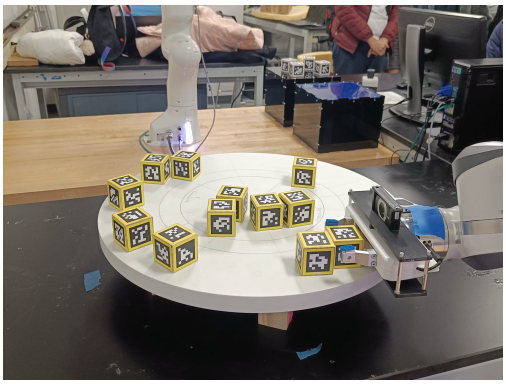
### 3.3.3 Lab Session 3 Test Cases and Results

During this lab, we took readings to align our simulation of dynamic block grasping with the lab environment.

We verified the dynamic scanning position, verified our 'waiting' position on the turntable, and selected a 4-second delay parameter between grasping attempts. It was also necessary to update the `exec_gripper_cmd()` parameters, as we observed that the robot occasionally failed to detect grasped dynamic blocks.



(a) Unsafe 'waiting' position



(b) Improved 'waiting' position

Figure 10: Third lab session progress

During this lab, we were able to grasp and stack all four static blocks and one dynamic block (see Appendix 1 for the best results of our code in simulation and hardware). However, the dynamic grasping was highly angle-dependent, which would prove problematic during the competition as discussed in the Analysis section.

# 4 Analysis

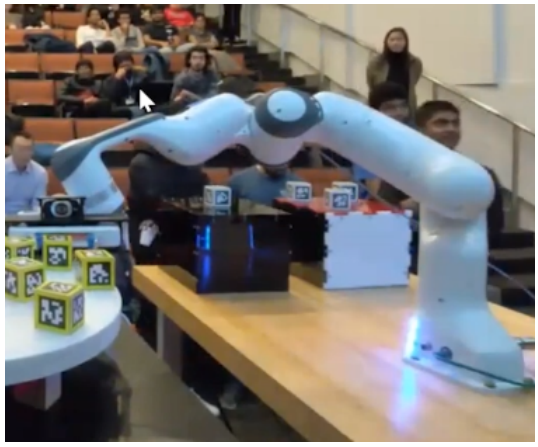
## 4.1 Code and Simulation Analysis

- In the first lab we found that our scanning position was not good which did not capture all the blocks due to shift in real world camera position which was later adjusted. Also the block position were given accurate according to the simulation which did not perform well during the hardware testing. These values were also adjusted for that.
- According to different strategy we had planned many stacking position but as we saw our progress in simulation and hardware testing, we kept moving towards optimum strategy.
- Also, the try-error implementation proved to useful for many simulations and hardware testings where we faced practical issues such as block not being actually picked up, camera reading not passing tolerance value etc.
- After lab 1, we also found out that the noise in camera readings were quite high and we reduced our tolerance accordingly. This tweak was still not sufficient in our competition as multiple static blocks were not registered by the camera. Initially we were also trying to obtain the gripping positions for dynamics blocks through simulation, but Ik solutions we obtained for the desired positions always touched the table which we did not desire. So decided to manually obtain the joint variables using lab0.

## 4.2 Competition Performance and Analysis

During our competition, we were consistently able to stack blocks in a round. However, there were two distinct issues that occurred – both with our dynamic code – which prevented us from stacking more.

(a) Round 1 Tower



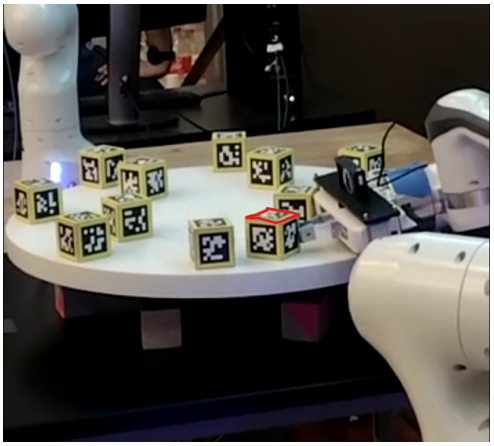
(b) Round 2 Tower

Figure 11: Static block stacking during competition

In our first match, we successfully stacked two static blocks on the blue side before attempting to acquire a dynamic block. The arm went to the pre-programmed pose at the turntable and closed at pre-determined intervals while the motion of the turntable brought blocks into range of the gripper at random angles.

As seen below, the first block to arrive was oriented at 45 degrees to the plane of the gripper. In consequence, when the effector closed, the block was pushed out without changing orientation. The effector tried to grip the block four times, as we expect it to do when it does not confirm that a block has been grasped, with similar results. Due to this issue, we were not able to grab a dynamic block, and the program terminated.

(a) Gripper attempts to close on 45-degree block



(b) Block is pushed without changing orientation

Figure 12: Failed dynamic grasping in competition due to block orientation

This is a risk we acknowledged when developing our dynamic block picking strategy, which we decided to proceed with due to the randomized nature of each match's block setup.
In addition, we encountered difficulties with the camera reading. There was more noise in the environment during the competition. Our value of error (epsilon) was not set sufficiently high to accommodate that level of noise. As a result, we experienced errors in the code during competition. More hardware testing or a trial in the competition environment would have helped us understand and rectify this issue.

## 4.3 Lessons Learned

- Progress on our code was not linear. We would frequently make progress early in a work session, partially solving an existing issues – for example IK accuracy – before becoming stuck on a new issue such as IK solution selection.
- Most of these moments of becoming 'unstuck' occurred as a result of performing hardware tests – for example, understanding the relationship between block H-matrix orientation and gripper approaches – or talking to other students. Going forward, it will be important to recognize when one's approach is not producing results, and how to find another using these methods.
- Scheduling in-person work times for a group of four during the end of the semester is difficult. As such, dividing up work early on or working ansynchronously may be more effective.
- Many method selections involve a tradeoff between repeatability and speed.
- Consider how obstacles will change as a result of your robot actions. When we selected a path-planning approach, we did not consider how a growing tower would act as an obstacle, leading to collision.
- While code segmentation – for example into `dynamicBlocksCenter` and `staticBlocksCenter` – allows for swifter debugging and better modularity of code, it is also important to consider how the segmented strategies will interact, as in the case of the static-to-dynamic transition that resulted in collision with the tower.

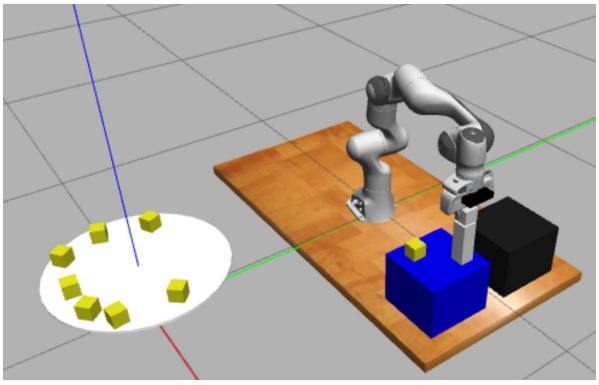
## 4.4 Future Directions

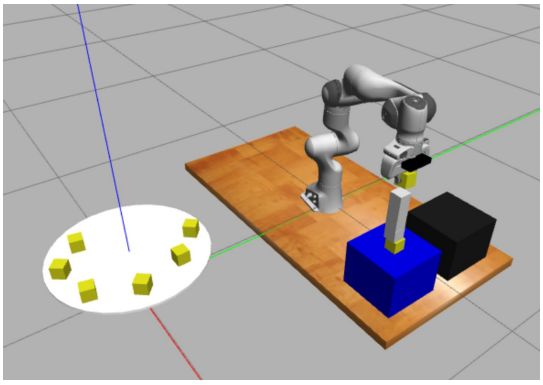
Based on the issues observed during competition, and the lessons learned, we would make a number of changes to our approach given more time or the opportunity to repeat this final project.

1. We chose to trade off speed of calculation against obstacle avoidance with respect to choice of path planning approach. Because the environment grows an additional 'obstacle' – the tower of blocks – during competition, this may not have been a considered trade. One option to address this limitation could be to make use of the potential planning method instead, with a dynamic list of obstacles that is updated with the position of tower blocks.
2. As described, our dynamic grasping strategy experiences variable success based on the angle at which the block enters the effector. During the completion, the angle was inopportune, so we were not able to successfully execute out our grasping routine. In order to make our dynamic strategy more robust, one option would be to add a slight side-to-side motion in the radial direction to encourage the block to adjust its angle, or a more complete overhaul of our dynamic strategy to incorporate the camera.
3. Improved handling of camera noise. We used a simple tolerance 'filter' to determine – based on the camera reporting – which axis of the block frame was initially pointing up or own. With more time, we would have been well to implement a more sophisticated signal processing approach such as a simple low-pass filter.

# 5 Appendix I: Successful Stacking in Simulation and Lab

## 5.1 Best Results in Simulation





## 5.2 Best Results in Hardware