MEAM 5200 - Final Project Report

Emily Paul, Harita Trivedi, Thomas Stephen Felix, Vaikhari Kharul

January 24, 2026

1 Methods

1.1 Static Blocks

Algorithm 1 Pick Up Static Block

```
1: Procedure PickUpStaticBlock()
2: $cubes \leftarrow$ GetCubesOnStaticBlock()                Find cubes on static block
3: if len($cubes$) = 0 then
4:     return False
5: end if
6: $cube \leftarrow cubes$.pop()                            Find robot pose to pickup a cube
7: $cube\_position \leftarrow cube[1:3,4]$
8: $planar\_angle \leftarrow$ GetPlanarAngle($cube$)
9: $pickup\_pose \leftarrow$ MakePickUpRobotPose($planar\_angle, cube\_position$)
10: $hovering\_pose \leftarrow$ MakeHoveringRobotPose($pickup\_pose$)
11: MoveToFrame($hovering\_pose$)                           Execute movement
12: OpenGripper()
13: MoveToFrame($pickup\_pose$)
14: CloseGripper()
15: MoveToDefaultStaticConfiguration()
16: return True
17: End Procedure
```

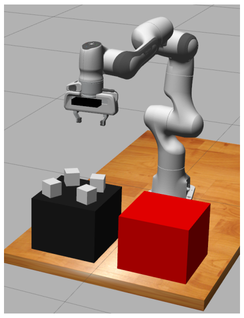
Algorithm 1 outlines the overall procedure on how a cube was identified and picked up on the static block. Lines 2-5 use object detection to find the cubes in the robot frame and ensure that there are sufficient cubes for the robot to interact with. Lines 6-10 find the relevant robot transforms required to pickup the cube and lines 11-15 execute the movement necessary to physically interact with the cube. We will inspect some components further :

- GetCubesOnStaticBlock

Algorithm 2 Get Cubes On State Block

```
1: Procedure GetCubesOnStaticBlock()
2: MoveToDefaultStaticConfiguration()
3: $cubes \leftarrow$ GetCubesInRobotFrame()        Find positions of cubes in robot frame
4: $cubes \leftarrow$ FilterCubesOnStaticBlock($cubes$)
5: return $cubes$
6: End Procedure
```

The function MoveToDefaultStaticConfiguration moves the robot to pre-defined joint configurations. These configurations are found when testing with the real robot that has the entire static platform within the FOV of the camera.


(a) red robot

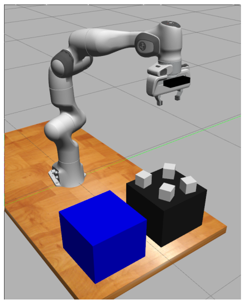

(b) blue robot

Figure 1: Default static joint configuration

Algorithm 3 Get Cubes In Robot Frame

```
1: Procedure GetCubesInRobotFrame()
2: $cubes\_in\_camera\_frame \leftarrow$ GetDetections()        Get cube detections from camera
3: $cubes\_in\_robot\_frame$ = Empty
4: for $H_{cube}^{camera} \in cubes\_in\_camera\_frame$ do
5:     $cubes\_in\_robot\_frame$.Append($H_{ee}^{robot} * H_{camera}^{ee} * H_{cube}^{camera}$)
6: end for
7: return $cubes\_in\_robot\_frame$
8: End Procedure
```

GetCubesInRobotFrame is described in algorithm 3. line 2 interacts with the physical or simulated camera to get the transforms of cubes detected by the camera. The cubes locations averaged over 10 detections before being returned. In lines 4-6, we calculate the position of each cube in the robot frame as

$$H_{cube}^{robot} = H_{ee}^{robot} * H_{camera}^{ee} * H_{cube}^{camera}$$

Finally FilterCubesOnStaticBlock uses certain boundry conditions for the static block (refer to 1) to filter out cubes not on top of the static platform. These are calculated from the dimensions provided of the environment.

	x_{min}	x_{max}	y_{min}	y_{max}	z_{min}	z_{max}
blue	0.437	0.687	0.044	0.294	0.2	0.3
red	0.437	0.687	-0.294	-0.044	0.2	0.3

Table 1: Boundry box for cubes on static platform

- GetPlanarAngle
  For a cube on the static platform, there will always be an axis pointing along the +ve z-direction of the robot frame. This can be identified from the 3rd row of the homogenous matrix. As an example

$$H_{cube}^{robot} = \begin{bmatrix} a & 0 & -b & x \\ b & 0 & a & y \\ 0 & 1 & 0 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Here the +ve y-axis of the cube is alighned with the z-axis of the robot frame. We use this information to calculate the angle the x-z plane of the cube makes with the x-y plan of the robot as

$$\alpha = \tan^{-1}(a/b)$$

We then limit this to $[-\pi/4, \pi/4]$ which is the minimum angle required to pick up a cube in any orientation.

- MAKEPICKUPROBOTPOSE and MAKEHOVERINGROBOTPOSE
  This methods create the hovering position and the pickup position. The hovering position, calculated in the robot frame is 0.025m (half the lenghth of the cube) above the pickup position. It ensures the robot will avoid collisions when attempting to pick up the cube by performing only a lateral movement in the z-direction.
  Let P_{cube} be the position of the cube in the world frame. In the transformation matrix calculate previously, this would refer to the first three rows of its last column.

$$R_{cube} = R_{vertical\_gripper} * R_{Z,\alpha}$$

  R_{cube} is the desired rotation of the end-effector in order to precisely pickup the cube.

$$R_{vertical\_gripper} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

  $R_{vertical\_gripper}$ is the rotation of the endeffector, such that the gripper is facing downwards to pick up the cube (The z-axis of the gipper is aligned with the -ve z-axis of the robot frame).

$$R_{Z,\alpha} = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

  $R_{Z,\alpha}$ is the rotation about the z-axis by planar angle α. We can now compose the hovering and pickup transforms in robot frame :

$$H_{pickup} = \begin{bmatrix} R_{cube} & P_{cube} \\ 0 & 1 \end{bmatrix}$$

  For the hover position, the hover offset is added to $H[3, 4]$ which refers to the position of the endeffector along the z-direction of the robot frame.

- MOVETOFRAME

Algorithm 4 Move To Frame

```
1: Procedure MOVETOFRAME(frame, robot_speed, configuration_bias)
2: $status, robot\_position \leftarrow$ IKPOSITIONNULL($frame, configuration\_bias$)
3: if $status = False$ then
4:     return $False$
5: end if
6: SETROBOTSPEED($robot\_speed$)
7: MOVETOPOSITION($robot\_poisition$)
8: return $True$
9: End Procedure
```

In algorithm 4, line 2 converts the target robot pose to be attained into joint configurations for the robot using Inverse Kinematics (IK). Line 6 & 7 will set the robot speed and move to the desired configuration.

To calculate the joint configuration, the IK solver uses a gradient descent method to solve the IK problem of the panda robot. We use 'J pseudo-inverse' as opposed to 'J transpose' as experimentation proved its better reliability. We use the current joint configuration of the robot as the seed (starting configuration) for the gradient descent solver. Finally, we add a vector in the nullspace to maintain a configuration bias. If no bias is given, the current configuration is used as the bias. This assumes that the solution required is close to the current configuration of the robot and helps to avoid wide, unnecessary and potentially dangerous movements.

$$b = (configuration\_bias - q)/(upper - lower)$$
$$dq = dq_{ik} + (I - J^{\dagger}J)b$$

1.2 Dynamic Blocks

Our strategy for dynamic blocks was to wait until a block enters a detection bracket, then predict forward its future pose for a given time horizon so that the arm can wait there at the appropriate position and orientation until the time elapses, at which point the block will be directly below the gripper. We then lower and close the gripper to capture the block.
Our detection bracket is a range of angles on the table, calling the positive x axis 0 and the positive y axis $\frac{\pi}{2}$, for reference.

Algorithm 5 Detect Block at Detection Angle

```
1:  Procedure BlockAtDetectionAngle($cube\_transforms, \beta$)
2:  $buffer \leftarrow \frac{\pi}{8}$                      Detection range buffer around the angle
3:  if IsEmpty($cube\_transforms$) then
4:      return False
5:  end if
6:  for $cube \in cube\_transforms$ do
7:      $\theta \leftarrow$ CalculateAngle($cube$)     Calculate cube's angle relative to the table center
8:      if $\beta - buffer < \theta < \beta + buffer$ then
9:          return $cube$                      Return the cube if it is within detection range
10:     end if
11: end for
12: return None                                No cube found within detection range
13: End Procedure
```

In algorithm 5, we check if there's a block in our detection bracket, consisting of our detection target angle β plus or minus a buffer, on the table.

- CalculateAngle
  We extract the x and y values from the input block pose and compute the angle on the table as $\theta = \arctan \frac{y}{x}$.

Algorithm 6 Predict Time to Catch Angle

```
1:  Procedure PredictTimeToCatchAngle($block\_pose, \gamma$)
2:  $\omega \leftarrow \frac{\pi}{60}$                                   Rotational speed of the table
3:  $\theta_{curr} \leftarrow$ CalculateAngle($block\_pose$)           Current block angle
4:  $t_{prediction} \leftarrow \frac{\gamma - \theta_{curr}}{\omega}$            Calculate time to reach catch angle
5:  return $t_{prediction}$                                Predicted time horizon
6:  End Procedure
```

In algorithm 6, we predict how long it will take the input block to move from its current angle on the table to the catch angle γ.

In algorithm 7, we first set the angular velocity of the rotating table, and then we compute the angle on the table at which the block is currently located. We then use the table's angular velocity to compute what this angle would be at the horizon that we're predicting for (where the arm will be

Algorithm 7 Predict Block Position and Angle at Horizon

```
1: Procedure PredictBlockPositionAngleAtHorizon($block\_pose, prediction\_time$)
2: $\omega \leftarrow \frac{\pi}{60}$                                                    Rotational speed of the table
3: $\theta_{curr} \leftarrow$ CalculateAngle($block\_pose$)
4: $\theta_{future} \leftarrow \theta_{curr} + \omega \cdot prediction\_time$                       Predict future angle of rotation
5: $radius \leftarrow$ CalculateRadius($block\_pose$)
6: $pos_{future} \leftarrow$ CalculatePosition($radius, \theta_{future}, block\_pose$)        Predict future block position
7: $rot_{future} \leftarrow$ UpdateRotationMatrix($\omega, prediction\_time, block\_pose$)
8: $\alpha_{future} \leftarrow$ GetPlanarAngle($rot_{future}$)                                Compute planar angle
9: $ee\_rot_{future} \leftarrow$ CalculateEndEffectorRotation($\alpha_{future}, R_{vertical\_gripper}$)
10: $future\_pose \leftarrow$ AssemblePose($ee\_rot_{future}, pos_{future}$)                  Build pose matrix
11: return $future\_pose, \theta_{future}$
12: End Procedure
```

waiting in a hover). We compute how far the the block is from the center of the table an use this effective radius, along with the predicted future angle, to compute the future position of the block. We then update the block's rotation matrix accordingly and extract its future planar angle to assemble and return the complete future pose.

- CalculateRadius
  We compute the distance of the block from the center of the table using the Pythagorean theorem, as $r = \sqrt{x^2 + y^2}$.

- CalculatePosition
  We compute the future positional values as follows:
  $x_{future} = radius \cdot \cos\theta_{future}$
  $y_{future} = radius \cdot \sin\theta_{future}$
  $z_{future} = z$

- UpdateRotationMatrix
  We use `scipy`'s Rotation library to construct a rotation matrix T corresponding to a rotation of $\omega \cdot prediction\_time$ about the z axis. We then extract the rotation matrix R from $block\_pose$ (the first three rows and columns) and compute the block's future rotation as $T \cdot R$.

- GetPlanarAngle
  Explained in section 1.1.

- CalculateEndEffectorRotation
  We use the Rotation library to construct a rotation matrix T corresponding to a rotation of $\alpha_{future} + \alpha_{offset}$ around the z axis, where α_{offset} is a hardware-specific adjustment that we gathered from tuning on the arms. We also pull the gripper's vertical rotation V, and compute the end effector's future rotation as $V \cdot T$.

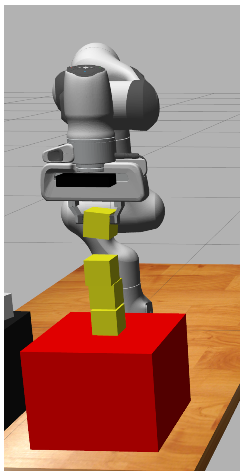
- AssemblePose
  We fill in the future pose's first three rows and columns with ee_rot_{future} and the last column with pos_{future}.

Algorithm 8 Pick Up Dynamic Block

```
1:  Procedure PickUpDynamicBlock()
2:  $\beta \leftarrow$ SetAngle(detection)
3:  $\gamma \leftarrow$ SetAngle(catch)
4:  MoveWithSpeed(DefaultDynamicPosition)          Move to default dynamic block position
5:  $start\_time \leftarrow$ CurrentTime()
6:  while True do
7:      $cubes \leftarrow$ GetCubeInTableFrame()                Retrieve cube positions on table
8:      $detection\_time \leftarrow$ CurrentTime()
9:      if NotEmpty($cubes$) then
10:         $cube\_tracked \leftarrow$ BlockAtDetectionAngle($cubes, \beta$)      Check detection bracket
11:         if $cube\_tracked \neq$ None then
                Break
12:         end if
13:     end if
14:     if CurrentTime() $-$ $start\_time \geq 120$ then
15:         End Procedure                  Exit if table fully rotated and no cubes were found
16:     end if
17: end while
18: $t_{catch} \leftarrow$ PredictTimeToCatchAngle($cube\_tracked, \gamma$)      Predict time to pickup
19: $[pos_{future}, \_] \leftarrow$ PredictBlockPositionAngleAtHorizon($cube\_tracked, t_{catch}$)
20: $[pos_{hover}, pos_{catch}] \leftarrow$ GetHoverCatchPositions($pos_{future}$)      Get configurations for pickup
21: $bias \leftarrow$ SetConfigurationBias()
22: MoveToFrame($pos_{hover}, 0.2, bias$)                Move to hover over pickup location
23: $time_{offset} \leftarrow$ SetTimeOffset()
24: OpenGripper()
25: while True do
26:     if CurrentTime() $-$ $detection\_time \geq t_{catch} - time_{offset}$ then
27:         $[status, \_] \leftarrow$ MoveToFrame($pos_{catch}, 0.3, bias$)
28:         CloseGripper()                      Lower and close gripper at pickup time
            Break
29:     end if
30: end while
31: MoveToPosition($pos_{hovering}$)                     Return to hovering position
32: End Procedure
```

Algorithm 8 is the function called in a loop in main once we finish stacking the static blocks, to keep stacking dynamic blocks until the rotating table is empty. When it is called, it first runs our block detection (algorithm 5) in a loop until we get a block to track. It then computes the time and location of pickup (algorithms 6 and 7 respectively) and moves the arm to the hover position. Then, once we hit pickup time (plus or minus some offset that we tuned on hardware), it drops the arm into the catch position (just a lower z value than the hover position) and closes the gripper to capture the block. Finally, it returns to the hovering position with the block. As shown in figure 2, this set of algorithms for detecting and picking up dynamic blocks, in conjunction with the stacking methods described below in section 1.3, is successfully able to stack dynamic blocks in simulation.

- GetHoverCatchPositions
  The hover and catch positions have the same x and y values as the block position we computed with algorithm 7, and their z values were manually tuned based on hardware testing.


(a) red robot

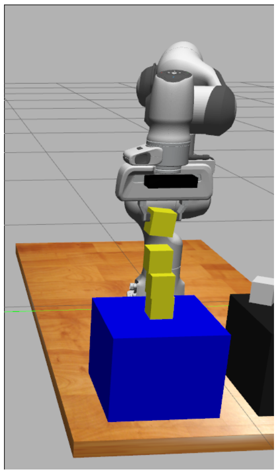

(b) blue robot

Figure 2: Dynamic block stacks

1.3 Stacking

[H] In algorithm 9, lines 3-12 find the placement and hovering robot pose. If no cube is found on the

Algorithm 9 Place Cube On Block

```
1: **Procedure** PlaceCubeOnBlock()
2: $cubes \leftarrow$ GetCubesOnPlacementBlock()          Find cubes on placement block
3: **if** len($cubes$) $= 0$ **then**
4:     $placement\_pose \leftarrow \begin{bmatrix} R_{vertical\_gripper} & P_{center} \\ 0 & 1 \end{bmatrix}$
5: **else**
6:     $cube \leftarrow$ SortByHeight($cubes$)          Find placement pose for new cube
7:     $P_{cube} \leftarrow cube[1:3,4]$
8:     $\alpha \leftarrow$ GetPlanarAngle($cube$)
9:     $O_g \leftarrow$ GetGripperFingerOffset($\alpha$)
10:    $placement\_pose \leftarrow$ MakePlacementRobotPose($P_{cube}, \alpha, O_g$)
11: **end if**
12: $hovering\_pose \leftarrow$ MakeHoveringRobotPose($placement\_pose$)
13: MoveToFrame($hovering\_pose$)          Execute movement
14: MoveToFrame($placement\_pose$)
15: OpenGripper()
16: MoveToDefaultPlacementConfiguration()
17: **End Procedure**
```

placement platform, the robot will attempt to place it on the center of the placement panel. If there are cubes on the placement platform, we find the highest cube's position and orientation to calculate the placement robot pose. This is to allow for variability in environment, unanticipated collisions or any other factors that may change the position of the blocks on the placement platform.

- GetCubesOnPlacementBlock
  Similar to GetCubesOnStaticBlock, we find the frame for cubes currently on top of the placement block and filter using a bounding box defined in 2.

- GetGripperFingerOffset
  When a cube is grabbed, it may not be in the center of the gripper. Let d_g be the deviation from the center along the direction of the gripper pins. To compensate for the offset we calculate the

	x_{min}	x_{max}	y_{min}	y_{max}	z_{min}	z_{max}
blue	0.437	0.687	-0.294	-0.044	0.2	0.9
red	0.437	0.687	0.044	0.294	0.2	0.9

Table 2: Boundry box for cubes on static platform

x-y offset to ensure the stacking is free of error as follows

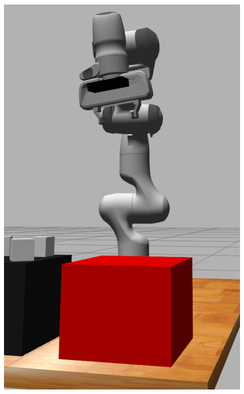
$$O_g = \begin{bmatrix} -d_g \sin(\alpha) & -d_g \cos(\alpha) & 0 \end{bmatrix}^T$$

- MakePlacementRobotPose and MakeHoveringRobotPose
  The rotation of the robot placement pose is calculated as follows

$$\begin{aligned} R_{place} &= R_{vertical\_gripper} * R_{Z,\alpha} \\ P_{place} &= P_{cube} + O_g + \begin{bmatrix} 0 & 0 & 0.025 \end{bmatrix} \\ H_{place} &= \begin{bmatrix} R_{place} & P_{place} \\ 0 & 1 \end{bmatrix} \end{aligned}$$

  For the placement pose, 0.025m is added along the +ve z-direction of the top cube. This is to place the cube on top of the highest cube. For hover position, an additional 0.025m is added.

- MoveToDefaultPlacementConfiguration: These are configuration pre-defines by testing with the hardware robot to ensure that the cubes on the placement platform are observable given the FOV of the real hardware camera.


(a) red robot

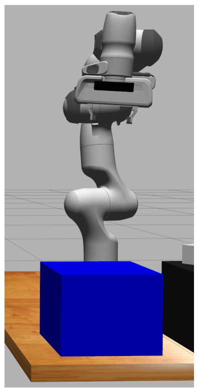

(b) blue robot

Figure 3: Default placement joint configuration

1.4 Overall Approach

Our strategy for the final project was multifaceted, focusing on both static and dynamic block manipulation. We developed one approach for static blocks and one for dynamic blocks, as detailed in previous sections. Our overall strategy was to:

- Begin early with static block manipulation, developing robust code for efficient stacking.
- Simultaneously work on dynamic block strategies, recognizing their increased complexity.
- During the testing phase, fine-tune static block stacking while rigorously testing dynamic block manipulation.

This approach allowed us to build a solid foundation with static blocks while progressively tackling the more challenging aspects of dynamic block handling.

2 Evaluation

2.1 Simulation

The performance of the blue and red robots was evaluated in simulation. Video demonstrations of the robots can be accessed at the following links:

- **Blue Robot:** Video Link
- **Red Robot:** Video Link

The robots' performance was evaluated based on accuracy in picking up and stacking blocks, as well as the time required for each step of the task.

For block-picking accuracy, the robots achieved 100% accuracy when handling static blocks and 82.75% for dynamic blocks. In terms of stacking, the robot demonstrated consistent performance with static blocks, successfully placing each block on top of the previous one. However, stacking dynamic blocks was less stable due to inconsistencies in the manner in which these blocks were picked up.

Time measurements were recorded for various steps, including cube detection, inverse kinematics (IK) calculations, movements to hover and pickup positions, gripper actions, and returning to a default position. Time statistics were averaged across the first four static blocks and the first four dynamic blocks.

- **Cube Detection:** Negligible time, averaging close to 0 seconds for both static and dynamic blocks.
- **IK Calculations:** Ranged from 0.2 to 2 seconds, depending on the complexity of the desired configuration. Static blocks required an average of 0.84 seconds, while dynamic blocks took longer, averaging 1.34 seconds.
- **Movement to Hover Position:** Averaged 3.78 seconds for static blocks and 5.61 seconds for dynamic blocks.
- **Movement to Pickup Position:** Static blocks averaged 5.23 seconds, while dynamic blocks required an average of 3.97 seconds.
- **Gripper Action:** Closing the gripper took an average of 0.25–0.35 seconds for static blocks. Dynamic blocks showed more variability, averaging 1.49 seconds.
- **Return to Default Position:** Averaged 2.23 seconds for static blocks and 5.20 seconds for dynamic blocks.

Trajectory execution took the longest time, ranging from 3 to 6 seconds. Additionally, waiting for a dynamic block to reach a specific angle before picking it up was a time-consuming step, particularly when fewer dynamic blocks remained.

Step	**Static Cube 1**	**Static Cube 2**	**Static Cube 3**	**Static Cube 4**	**Static Avg.**
Cube Detection	0.001	0.003	0.0	0.0	0.001
IK (Avg.)	0.96	0.88	0.74	0.77	0.84
Moving to Hover	4.45	3.56	3.62	3.48	3.78
Moving to Pickup	5.35	5.14	5.12	5.33	5.23
Closing Gripper	0.28	0.27	0.30	0.27	0.28
Default Position	2.14	2.30	2.18	2.30	2.23

Table 3: Time statistics for static blocks.

Step	Dynamic Cube 1	Dynamic Cube 2	Dynamic Cube 3	Dynamic Cube 4	Dynamic Avg.
Cube Detection	0.0	0.0	0.0	0.0	0.0
IK (Avg.)	1.20	1.28	1.34	1.54	1.34
Moving to Hover	5.20	5.30	5.87	6.08	5.61
Moving to Pickup	3.53	3.58	4.38	4.38	3.97
Closing Gripper	0.32	0.29	0.31	5.03	1.49
Default Position	5.03	5.63	5.02	5.12	5.20

Table 4: Time statistics for dynamic blocks.

2.2 Hardware

During hardware testing, we had two main objectives.
First and foremost, we had to to fix our gripper commands, which worked as expected in simulation but were very inconsistent on hardware. Specifically, once closed around a block, the `open_gripper()` command ceased to execute, and our program would simply wait in whatever configuration was last commanded until the gripper timed out and released. We removed the state checks we had put in regarding the width of the gripper after the `close_gripper()` command executed, to verify whether it had a block in its grasp, but the issue persisted. We finally were able to work around this by manually opening and closing the gripper with the `exec_gripper_cmd`.
Secondly, we tuned manual offsets for the z axis positions in the hovering and catching configurations and the timing of the transition between the two (both described in section 1.2). We also made minute adjustments in the xy plane for each arm before the competition.
One issue that we discovered when testing on hardware was that the block detection sometimes happened from the top face of the block and sometimes from one of the side faces. This did create issues for our catch positions, because our code did not account for this when picking z values, it simply assumed that the top face of the block was detected (because that is how it consistently ran in simulation) and used the block pose z value accordingly. As a result, in cases when a side face was detected instead of the top face during hardware tests, the catch position would be assigned too low, and the gripper would collide with the table as a result. We did not have time to correct this issue before the final hardware run, but the solution would have been to run a check on the z value in the detected block pose and adjust the catch position accordingly.

3 Analysis

Overall, there were more challenges in the system when faced with the dynamic blocks. This may be due to the limited field of view of the end effector camera, time delays caused by the IK calculations, and just in general challenges that come with adjusting the movement of the robot to a moving block.

This system is quite robust when it comes to picking up the static blocks and stacking them. Even when the state of the stacking block changed, the robot was able to adapt to the new tower and stack accordingly. With more time, default position for dynamic block picking could have been optimized to allow for more success in terms of picking up the dynamic blocks.

The simulation to hardware gap was difficult to bridge. Even though the static block picking was working in simulation, it was challenging to adapt it to hardware due to April Tag glare, Z-axis orientation differences, and offsets in the system.

This kind of stacking algorithm can be useful in robotic tasks that involve assembly or just logistics in general. It relies on repetitive motions through inverse kinematics, which means it would work well for simple environments where there are not many obstacles in the workspace. This system would have difficulty adapting to different types of objects or a more cluttered environment.

4 Lessons Learned

4.1 Static Block Handling

- We realized the importance of accounting for potential stack instability. We implemented a strategy where the Franka Panda assesses the topmost layer of the stack before placing another block, enhancing overall stability.

- Initially, we considered building separate stacks for static and dynamic blocks. However, we reconsidered this plan after feedback during our presentation, recognizing that it could lead to more complex path planning and increased risk of collisions.

4.2 Hardware Calibration and Adaptation

- During our second week of open labs, we learned the critical importance of calibrating for hardware offsets, which differed from simulation.
- We observed overshooting issues in hardware stacking, even when the code performed well in simulation. This taught us the necessity of accounting for Z-axis offsets specifically in hardware.

4.3 Dynamic Block Strategies

- Timing was a significant challenge in positioning the arm to catch dynamic blocks after sensing their location and orientation.
- We experimented with two approaches:
  1. A vertical approach similar to static blocks.
  2. A horizontal sweeping strategy.
- The sweeping strategy proved unreliable, often displacing blocks. We ultimately chose the vertical approach for its reliability and robustness, though it required careful timing adjustments.

4.4 Performance Optimization

- To improve overall task speed, we increased joint velocities after realizing that static block stacking alone consumed half of the allotted competition time.

4.5 Competition Environment Challenges

- The final competition environment presented new challenges, including different offsets that we couldn't fully account for due to time constraints.
- Increased gripper pressure sensor sensitivity in the competition setup required more precise block centering to avoid self-collision detection, a crucial last-minute learning experience.

These lessons underscore the importance of adaptability, thorough testing in various conditions, and the need to anticipate and prepare for hardware-specific challenges when transitioning from simulation to real-world environments.