MEAM 5200 Final Project Report (Team 1)

Brandon Y. Yang
yang52@seas.upenn.edu
Bangan Wang
wangban2@seas.upenn.edu
Zihao Zhou
zihao21@seas.upenn.edu
Sunny Yu
sunny.yu@seas.upenn.edu

1 Introduction

The MEAM5200 Final Project is a pick-and-place task in which two teams control Franka Panda manipulators in a shared environment (Fig. 1) to acquire and stack wooden blocks. The objective is to maximize the total score within fixed time duration. Score is determined only by the final block configuration at the end of the match: stacking blocks higher yields more points, and dynamic blocks are worth twice as much as static blocks at the same height. Accordingly, team's goal is to execute reliable block acquisition and tower-building actions that maximize the final score under sensing noise, motion uncertainty, and time constraints.

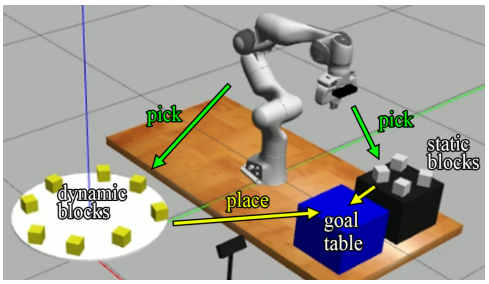

Figure 1: Simulated final project environment. (click here to view our controller's performance)

The scope of this work focuses on grasping and placing blocks, as well as designing a decision-making layer that maximizes score. We do not consider adversarial strategies such as interfering with the opposing robot, nor do we aim to optimize raw motion execution speed. Instead, this work presents and validates an end-to-end block-stacking framework composed of **three** core components: perception, high-level control (*think*), and low-level execution (*act*).

At the **perception** layer, we use an end-effector-mounted camera with denoising strategies to detect blocks and maintain an internal world state, including block availability and the current tower configuration. This layer addresses the challenge of converting noisy, partial observations under random block placement into a consistent representation suitable for planning.

At the **high-level control** layer, we implement a finite-state machine (FSM) that maps the evolving world state into score-seeking manipulation decisions. This includes selecting which block source to target, sequencing pick-and-place actions, and invoking recovery behaviors when failures occur, enabling adaptable task execution.

At the **low-level execution** layer, we translate high-level policies into safety-constrained robot commands through a collection of low-level controllers. This layer bridges the policy-to-command gap by producing feasible and repeatable motions that respect system limits and operate consistently in both simulation and on hardware.

Experiments are conducted in both simulation and on physical hardware to validate the proposed framework. Performance is evaluated primarily by the final score at the four-minute mark, with additional metrics including grasp and transfer success rates and failure-recovery effectiveness. Detailed evaluation metrics are introduced in the corresponding experimental sections.

2 Method

2.1 High-level Controller (FSM)

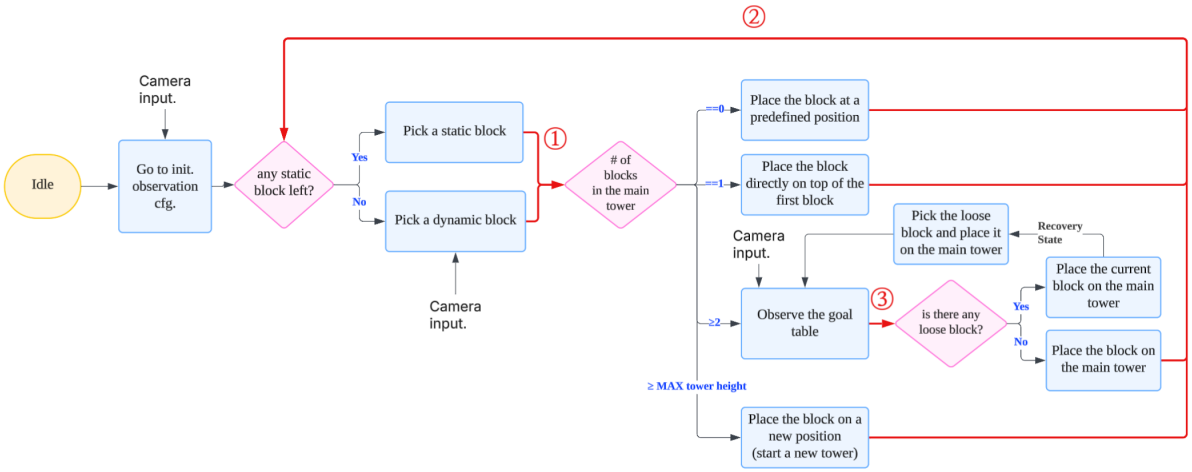

Figure 2: High-level diagram of FSM. The red arrows represent updating the *world state*

The overall structure of the high-level FSM is shown in (Fig. 2). When evaluating execution conditions, the entire system relies on the current *world state*. We model the environment world state as a collection of **three** dictionaries, including *{picked static blocks, blocks in the main tower on the goal table, loose blocks on the goal table}*. The system starts from the idle state and first moves to a pre-defined observation configuration above the static-block table to read the poses of all static blocks in one shot. It then picks the block with the smallest norm distance to the current configuration. After each static-block pick, the picked-static-block dictionary is updated. Only when no static blocks remain does the system switch to picking dynamic blocks.

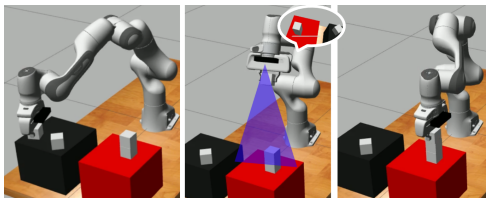

Figure 3: The robot observe the goal table before placing the block

After picking, the robot needs to place the block. During placement, the number of blocks currently in the main tower on the goal table determines the placement strategy. If the number of blocks in the main tower equals 0, it means there are no blocks on the goal table yet, so the block is placed at a hard-coded position; if the number equals 1, it means there is already one block, so the current block is placed directly on top of it. However, as the tower grows, relying on a fixed hard-coded increment becomes risky. Therefore, starting from the third block, before each placement we re-observe the goal table to update the world state (Fig. 3): how many blocks are in the tower and how many are loose blocks (if a single block appears next to a stack, we define it as a loose block). If there is no loose block, we read the (x, y) positions of all blocks in the tower, take the average to obtain the centroid, and compute a weighted combination of this centroid and the hard-coded nominal position to generate the next placement location; the robot then places the held block at this semi-hardcoded position (while the z position remains hard-coded). If there is a loose block, the robot places the current block using the same method, then prioritizes picking the loose block and stacking it onto the tower as a recovery mechanism to handle unexpected collapses. After each placement, the world state is updated and the loop repeats.

$$(x, y)_{\text{placement}} = w \cdot (x, y)_{\text{observed centroid}} + (1 - w) \cdot (x, y)_{\text{hard-coded}} \tag{1}$$

A special design choice is that once the main-tower height exceeds a threshold (`MAX_tower_height`), further stacking is treated as high-risk. The robot then selects a new placement location from a predefined list of candidate positions to start a **new** tower. After that, subsequent blocks are placed on this new tower, and the previous tower is treated as "complete". To identify which stack is the main tower, we use the rule that blocks whose (x, y) coordinates lie within a threshold are considered stacked together as one tower; if that tower's height is below the maximum tower height, it is considered incomplete and will be further built in subsequent steps.

2.2 Perception Module

In the FSM, each time the robot observes block poses to update the world state, it relies on the camera input. Block poses are estimated via AprilTag detection in camera frame $\{C\}$. The detector provides transformations T_b^c for each detected block. The block pose in robot base frame $\{0\}$ is computed via the transform chain, $T_b^0 = T_e^0(q) \cdot T_c^e \cdot T_b^c$ (where T_c^e is the fixed camera-to-end-effector calibration transform).

In simulation, the detector provides perfect transformations T_b^c for each block without noise. However, this is not the case for AprilTag detection on real hardware. To mitigate sensor noise and detection jitter, pose estimates are refined using temporal averaging over a sliding window. For each block b, N_{window} consecutive measurements $\{T_b^0[k]\}_{k=1}^{N_{\text{window}}}$ are collected at a fixed viewing configuration q_{view}, an **initial mean** of all these readings is calculated. To further improve the reading robustness, an additional outlier rejection is performed using the Frobenius norm as a distance metric. For each measurement $T_b^0[k]$, the deviation from the **initial mean** is quantified as:

$$d_k = \left\| T_b^0[k] - \bar{T}_b^0 \right\|_F = \sqrt{\sum_{i,j} \left(T_b^0[k]_{ij} - \bar{T}_{b\,ij}^0 \right)^2} \tag{2}$$

Measurements satisfying $d_k < \tau_{\text{outlier}}$ are retained. The final filtered reading is obtained by averaging these refined values.

2.3 Low-level Controllers

2.3.1 Block Grasping Controller

To grasp a block, we first need to align the end effector's (x_e, y_e) axes to the block. Blocks may have arbitrary planar orientation on the table surface. To compute a grasp pose, we first identify the block's horizontal axes, which are parallel to the global xy-plane (same as robot base xy-plane).

Let $R_b^0 \in SO(3)$ denote the rotation component of T_b^0, with columns $[x_b, y_b, z_b]$ representing the block frame axes in robot base coordinates. For each axis $u_i \in \{x_b, y_b, z_b\}$, compute its projection onto the xy-plane:

$$u_{i,\text{flat}} = [u_{i,x},\, u_{i,y},\, 0]^\top, \quad n_i = \|u_{i,\text{flat}}\|_2 \tag{3}$$

The axis with maximum planar projection is selected, and then two orthonormal horizontal axes are constructed as:

$$u_1 = \frac{v_{i^*,\text{flat}}}{n_{i^*}}, \quad u_2 = \frac{\hat{z} \times u_1}{\|\hat{z} \times u_1\|_2} \tag{4}$$

where $\hat{z} = [0, 0, 1]^\top$ is the vertical axis.

To align the end effector with the block, **four** candidate grasp orientations are available (Fig. 4). For each mode, IK is attempted in sequence. The first available IK solution will be deployed to the robot.

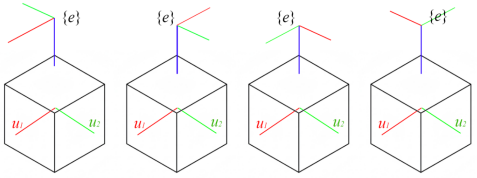

Figure 4: Four different alignment mode

One of the rationale behind this is **grasp stability**. Aligning the parallel-jaw gripper with the block's flat faces maximizes contact area and friction, avoiding the slip associated with corner grasps. Second, it leverages **kinematic redundancy**. While the four orientations ($\pm u_1, \pm u_2$) yield the same physical grasp, they require different joint configurations. Iterating through these options increases the likelihood of finding a valid IK solution.

After obtaining the alignment pose, the robot simply moves to a fixed position above the block, lowers the gripper, closes the gripper, and lifts to complete the grasp. This controller serves as the baseline for both static and dynamic block grasping.

2.3.2 Dynamic Block Grasping Controller

The dynamic-block grasping procedure consists of five main steps. The robot first moves to a fixed observation pose above the turntable. It then estimates the current angular velocity of the turntable and uses it to predict the target block pose after a fixed time horizon. Next, the robot moves to the predicted pose and grasps the block, and finally transfers it to the goal table.

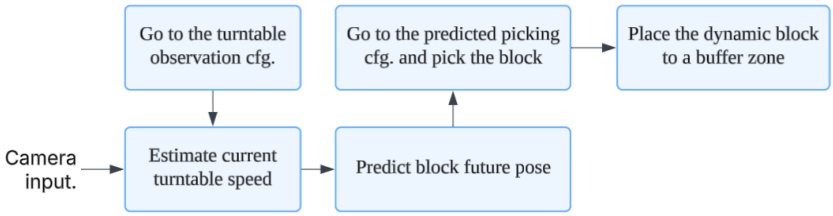

Figure 5: Dynamic controller architecture

Turntable Velocity Estimation: The turntable angular velocity ω is estimated via linear regression on observed angular positions. For **each** block b on the turntable in the current field of view, its angular position at time t is computed in the turntable frame $\{\tau\}$ as $\theta_b(t) = \arctan 2(y_b^\tau(t),\, x_b^\tau(t))$, where $(x_b^\tau(t), y_b^\tau(t))$ denotes the planar coordinates of block b expressed in $\{\tau\}$. The turntable frame origin coincides with the center of the turntable and shares the same z-axis as the robot base frame.

When multiple blocks are visible in FOV, individual velocity estimates are computed per block. Outliers are rejected using median absolute deviation, and the final estimate is the mean of inliers, similar to our prior method of noise reduction when detection blocks' pose.

We chose regression over averaging instantaneous velocities ($\Delta\theta/\Delta t$) to handle sensor noise. Differentiating noisy position data amplifies high frequency jitter, which results in unstable estimates. Regression uses the entire window of data to fit the underlying trend, which is another form of performing a smoothing operation.

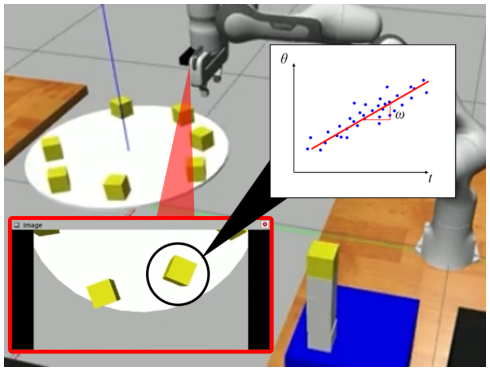

Figure 6: Turntable velocity estimator

Block Pose Prediction: The robot should decide which block to pick in FOV. When multiple dynamic blocks are visible, the robot selects the block that will reach the picking region the latest, giving more time to solve IK. Concretely, for the red team we choose the visible block with the minimum x-coordinate, while for the blue team we choose the visible block with the maximum x-coordinate.

Given target block's initial pose $p_b^\tau(t_0) = [x_0, y_0, z_0]^\top$, the robot will utilize the turntable velocity to predict its future pose after a fixed time interval. The **position** at $(t_0 + \Delta t)$ is simply estimated by first converting block's initial position in turntable frame to polar coordinates, advancing the angular component by $\omega \Delta t$, then converted back to (x, y) coordinate in $\{\tau\}$. The **orientation** estimation first computes the angle α between the block's midline at its initial position and the turntable tangent direction (clamped to an acute angle). Since a dynamic block does not move relative to the turntable, this angle remains constant after Δt. Therefore, we can use the predicted position to determine the tangent direction at the predicted location, and then recover the block's orientation accordingly (Fig. 7).

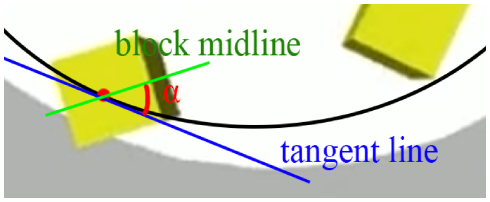

Figure 7: Orientation estimator

Timing-Based Grasp Execution: The overall dynamic-grasp procedure is mapped onto a timeline (Fig. 8). At time t, the robot first records the initial pose and uses it together with the estimated turntable angular velocity to predict the block pose at $t + \Delta t$, followed by IK solving. To avoid collisions with non-target blocks while waiting at the grasp pose (which could disturb the prediction), the arm waits at a pre-grasp configuration with a fixed height above the turntable until the preceding block clears the grasp zone. Upon reaching this pre-grasp configuration, we record the time t_{pre} to compute the remaining time before the predicted grasp moment. If this remaining time is negative, the robot can no longer catch the block and the procedure restarts. Otherwise, the remaining time is scaled by a weighting factor w and executed via `rospy.sleep`. After the wait, the robot lowers the gripper and immediately closes it to complete the grasp (can check the video in Fig. 1 for better interpretation).

Post-grasping block handling: Since we cannot be as confident as in the static case that the dynamic block is centered in the gripper, placing a potentially offset block directly onto the tower could destabilize the existing stack. Therefore, after grasping a dynamic block, we first "drop" it to a corner of the goal table and then re-observe the goal table. This allows the robot to re-grasp the dynamic block using the

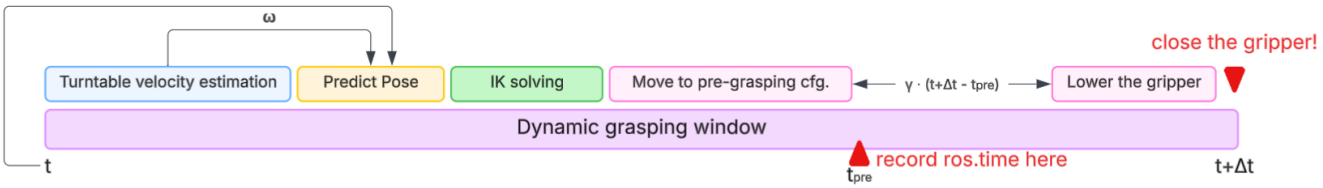

Figure 8: Timing-based grasp execution

same pipeline as for static blocks and place it on top of the tower without the offset issue.

2.3.3 Block Placement Controller

After picking a static or dynamic block, we need to place it to the top of the tower. For simplicity, we didn't implement any path planning algorithm such as RRT or APF. To prevent the carried block from inadvertently contacting the existing tower during transport and placement, we adopt a side-approach stacking strategy (Fig. 9). Specifically, the robot first moves to a lateral offset pose beside the tower, then transitions to a pre-place pose directly above the tower, and finally descends vertically for placement. The lateral approach direction is computed from the difference between the block lift location and the tower base (x, y) location, ensuring the end-effector approaches from the safer side rather than sweeping across the tower.

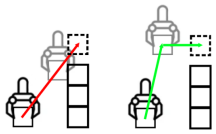

Figure 9: Side-approach stacking strategy

It's crucial to ensure the smoothness of the landing process of a block. A fast landing with high impact could cause lower-level blocks to slide, rotate, oscillate etc., deviating from their original placement location and orientation, changing the centroid location of the tower, possibly causing upper-level blocks to collapse. Thus, we use an impedance control strategy and generate a smooth vertical trajectory from the pre-placement configuration q_{above} to the target descent height z_{target} using a cubic polynomial profile:

$$s(t) = 3\left(\frac{t}{T_{\text{descent}}}\right)^2 - 2\left(\frac{t}{T_{\text{descent}}}\right)^3, \quad t \in [0, T_{\text{descent}}] \tag{5}$$

The commanded configuration and velocity at each timestep are:

$$q_{\text{cmd}}(t) = q_{\text{above}} + s(t)\left(q_{\text{target}} - q_{\text{above}}\right) \tag{6}$$

$$\dot{q}_{\text{cmd}}(t) = \frac{s(t+\Delta t) - s(t)}{\Delta t}\left(q_{\text{target}} - q_{\text{above}}\right) \tag{7}$$

The smooth velocity profile reduces abrupt velocity changes and minimizes impact on the stacked structure.

3 Evaluation & Analysis

We evaluate our approach on both the provided simulator and a real Franka robot. Due to limited hardware accessibility, most testing of the control logic was carried out in simulation. Hardware experiments instead focused on tuning a small set of parameters that are critical in the real system but show little effect in simulation (e.g., camera filtering parameters).

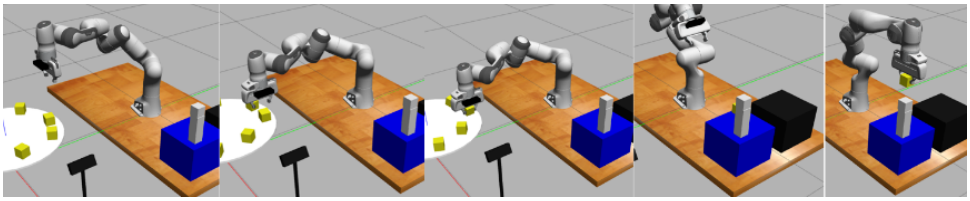

Figure 10: Dynamic mode logic and task sequence verified in simulation

3.1 Simulation Testing

3.1.1 Full-Program Run-Through & Overall Performance Evaluation

Setup: To verify the high-level logic of FSM, we first run through the full program in the simulator 20 times. Each trial run begins with 4 static blocks randomly placed in the source region and 8 dynamic blocks on the turntable, and the program is executed without interference until 3 dynamic blocks are stacked on the goal table. 15 of the trials set max tower height = 5, the remaining trials set max tower height = 4. We check if FSM correctly executes the planned sequence of tasks following the correct logic as described in (Fig. 2). Key metrics includes **block-grasping success rate, block-placement success rate, cycle run-time, and resulting tower height within the set time limit**. We record significant flaws if any, including collision, unsafe positions, unwanted response to failure of intermediate tasks (such as immobilization or random movements) etc.

Results: In every single trial, the FSM executed the exact sequence of planned tasks following the correct decision logic without any noticeable flaws. Specifically, it correctly performs in planned actions in chronological order as discussed in 2.1. Across all trials, the high-level decision logic of the FSM exhibited **zero** error. Furthermore, there was no collision, no unsafe pose, unwanted failure response or any other significant flaws, suggesting that the code logic is sound and complete, and our geometry calculations are correct.

Corner Cases: In addition to run-through trials, we specifically tested the FSM's response to the corner case of a tower collapsing due to failed placement. In these tests, we manually re-locate the top 1 or 2 blocks in the main tower to the table surface and rotate them into arbitrary orientations. We observe that the FSM correctly identifies the presence of these stray blocks and successfully re-stack them onto the main tower with good accuracy before proceeding to grab the next available block. In another test, we deliberately placed the tower in the collision zone and let the full tower collapse as the robot approaches. In this case, the FSM is still able to re-assemble all scattered blocks into a standing main tower. In our simulation trials, the FSM successfully rebuilds collapsed towers 100% of the times. The case of IK failure is also tested. The FSM correctly abandons the current grasp attempt and returns to the observation pose to identify an alternative target. The robot doesn't freeze or move unexpectedly. In summary, the FSM logic sensibly automates all tasks required for the project goals. Simulation results verify that the FSM takes into account all possible configuration types on the goal table, anticipates the full range of circumstances, including corner cases, that could arise during the grasping and placement of target blocks, and issues the appropriate response action accordingly 100% of the times, therefore in terms of high-level logic our program is a complete solution that successfully addresses the full scope of the project.

Global Performance Data: Across the 20 run-through trials, the robot arm executed in total 80 and 84 moves to pick static and dynamic blocks respectively. Among them, all 80 static blocks were successfully picked up while 60 dynamic blocks were successfully picked up, corresponding to a success rate of about 70%. Every block that was successfully picked-up was then successfully placed at its planned position on the goal table. Placement success rate is 100%. No trial reached the max tower height in less than 3 minutes. Around 80% of the trials reached the max tower height within the 5-minute time allotted,

depending on whether the first dynamic grasp is successful, and the average time to complete one pick-and-place action is found to be 20 seconds for static blocks and 46 seconds for dynamic blocks. These results show that the low-level execution of planed tasks is reasonably robust, while there is still room for improvement in dynamic block grasping. Improvement strategies are discussed in the next sections.

Table 1: Performance Statistics in Simulation

Metric	**Result**
FSM Task Success Rate	100.0%
Static Grasp Success	100.0% (80/80)
Dynamic Grasp Success	70.0% (60/84)
Placement Success	100%
Avg. Cycle Time	20 s/static block, 46 s/dynamic block

3.1.2 Parameter Tuning

After verifying the correctness of the code structure, we proceed to tune the values of important parameters. All parameters used in the code and their final values are listed in Appendix A.

Velocity Estimation Window: As discussed in 2.3.2, before grasping dynamic blocks, the robot will estimates the turntable speed by observing the positions of blocks within current FOV over a time window of length T_{window} seconds. The length of the window can affect of the estimation accuracy, especially in the real world where camera noise is large. We vary the T_{window} and observe the resulting estimation accuracy in simulator first. Our goal is to find the smallest T_{window} that produces sufficient accuracy. The data (Fig. 11) are plotted and a convergence toward the true turn-table speed and reduction of variance is observed as the time window is increased. We settled for $T_{\text{window}} = 4$ during competition since further lengthening the time window no longer improves the mean estimated velocity noticeably.

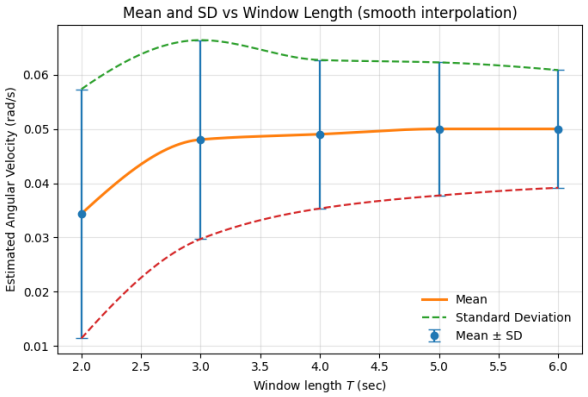

Figure 11: Estimated Angular Velocity vs. Length of Observation Window

Impedance Descent Parameters: As discussed in 2.3.3, to build a stable tower, we must ensure every block is placed onto the existing tower in a soft and controlled manner. Here, a desired end-effector trajectory given by parametric equation is prescribed, and a number of discrete points along the trajectory are along with the velocities at these way-points are fed into the provided impedance controller to track the trajectory. Here we test various descent trajectories $s(t)$ given by different types of functions, including **polynomials of various orders, trigonometric, and exponential functions**, all designed to minimize the block's impact velocity upon landing. We then evaluate the performance of the impedance controller at tracking the desired trajectory, particularly the influence of the number of discretize way-points on

the quality of the trajectory tracking. We consider the following trajectory functions, all approach zero theoretical velocity as the block descends towards the tower.

Cubic Polynomial
$$s(t) = 3t^2 - 2t^3$$

Quintic Polynomial
$$s(t) = 10t^3 - 15t^4 + 6t^5$$

Septic Polynomial
$$s(t) = 35t^4 - 84t^5 + 70t^6 - 20t^7$$

Raised Cosine
$$s(t) = \tfrac{1}{2}(1 - \cos(\pi t))$$

Exponential S-curve
$$s(t) = \frac{1}{1 + e^{-k(t-0.5)}}$$

We invoke the impedance controller to track these descent trajectories in simulation using **150** discretized way-points. The position vs. time graph (Fig. 12) shows that the quintic polynomial is least smooth, while trigonometric and cubic are most smooth. This is further verified by computing the integral smoothing-spline score J for each trajectories (The J score for each trajectory is tabulated in Appendix C). In addition, Cubic Polynomial and Exponential descent trajectories are faster than the rest.(see figure 12)

$$J = \int_{t_0}^{t_f} \left(\frac{d^2 s(t)}{dt^2} \right)^2 dt$$

We settled for cubic polynomial due to its smoothness, speed, and simplicity. We then observe the quality of the trajectory traced when we vary the **number of control way-points**. It is apparent from the data that fewer way points correspond to less high-frequency oscillation and hence lower smoothing spline score. However with fewer way-points the trajectory is more fragmented with more distinct jumps in velocity. (Fig.13) Both high frequency oscillations and discrete jumps have negative impact on block placement. We identify a good balance between $N = 100$ and $N = 150$. We further verify that the in this N range, the resulting end-effector velocity in the last time-step approaching the tower is vanishingly small, approaching zero just as we desire. (see end effector speed in the last descent step in Appendix D).

A possible explanation is that using a large number of way-points decreases the time step between updates, which amplifies numerical differentiation noise and induces large apparent accelerations, which in turn excites high-frequency jitter in the impedance-controlled response and increases interaction forces. Moreover, the robot generate continuous joint motion segment-by-segment between way-points; when segments become too short, the trajectory becomes more piecewise and less smoothly connected, which further increases jitter. Using fewer way-points increases the time spacing between reference updates, resulting in a more piecewise velocity profile.

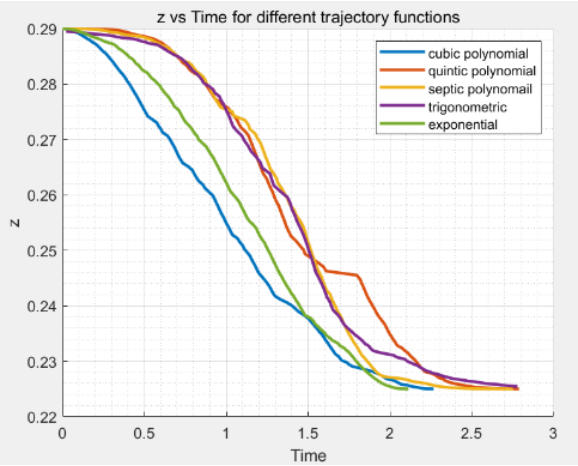

Figure 12: Position vs. Time during Impedance Descent for different trajectory functions (150 control way-points); End effector speed for the last descent step in Appendix D.

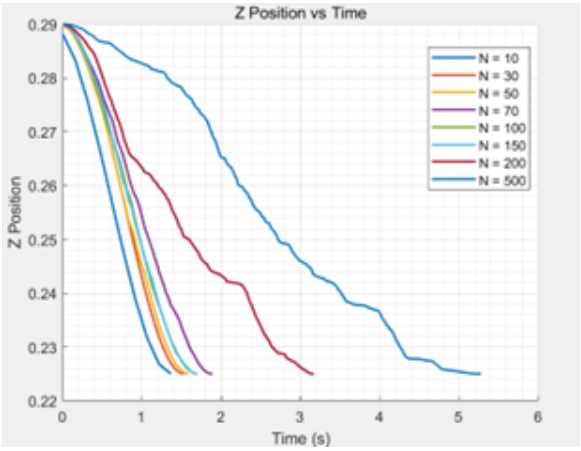

Figure 13: Position vs. Time during Impedance Descent for different numbers of control way-points N (cubic polynomial trajectory); End effector speed for the last descent step in Appendix D.

Dynamic Wait Constant: As discussed in 2.3.2, the robot estimates the turn-table speed and predicts the pose of the target block 20 seconds into the future ($\Delta t = 20$). The gripper is then moved to that predicted pose and waits until the predicted arrival time of the block to lower the gripper and grasp the moving block just as it passes under the waiting gripper. However, the gripper takes some time to lower and to account for that time, we define a wait-constant γ_{wait} (Fig. 8). We denote the ros time the robot arrives at the waiting position to be t_{pre}, and command it to wait for $\gamma_{\text{wait}}(t + \Delta t - t_{\text{pre}})$ seconds before lowering and closing the gripper. We tuned γ_{wait} primarily through iterative trial-and-error. Specifically, we tested values in the range $[0.2, , 0.9]$ and observed grasp outcomes; when a setting failed, we qualitatively attributed the failure to the gripper being lowered too early or too late. We finally selected $\gamma_{\text{wait}} = 0.5$, which yielded the most reliable performance in our trials (up to a 70% grasp success rate as discussed in Table 1).

In reflection, using a constant to γ_{wait} to control the gripper descent time is not optimal because while the gripper takes a fixed amount of time t_g reach grasping height in each trial, t_{pre} is different for each run, because each target block will be at a different pose 20 seconds into the future thus it takes a different time t_{pre} for robot to solve IK and move to the wait position. A much better strategy is simply to measure t_g and directly subtract set the wait time $t + 20 - t_{\text{pre}}$ is always accurate (where t is the ros time at the start of dynamic picking process).

3.2 Hardware testing

3.2.1 Camera Offset Calibration

The real robot and the simulator use different camera offsets, which makes the raw ${}^{c}T_b$ returned by the detector helper function inaccurate if used directly. Therefore, for each robot, we apply a fixed compensation offset to the (x, y, z) translation of the detected ${}^{c}T_b$. These offsets were obtained empirically via enumeration and repeated trials on the physical system. Specifically, we compared the gripper's actual reached (x, y, z) position with the measured block-center (x, y, z) position; when the absolute difference in each axis was below 0.5 cm, the offset parameters were considered reliable. The calibrated offsets for each robot are listed in Appendix B.

3.2.2 Block Placing Weighting Factor

As mentioned in Section 2.1, the placement location for each block is computed as a weighted combination of the observed tower centroid and a hard-coded tower base, controlled by weight w. When $w = 0$, the placement is fully determined by the hard-coded tower base; when $w = 1$, it is fully determined by the observed centroid. However, camera observation on hardware could degrade centroid-based placement due to external noises. Therefore, on hardware we tested $w \in \{0, 0.25, 0.5, 0.75, 1\}$ by stacking **three** static blocks under the same setting and measuring the horizontal span between the leftmost and rightmost edges of the stack (Fig. 14). Each setting was repeated three times and averaged. A span closer to 5 cm indicates better alignment ($-$ means no success trials to stack 3 blocks).

The results do not follow a strictly linear trend, as the distance span is influenced by factors beyond the weighting factor itself, such as grasp centering errors caused by AprilTag pose inaccuracies. Ideally, the block is centered in the gripper at grasp time; in practice, small offsets can occur, so even under the same w the outcomes can vary slightly across trials. Nevertheless, the data suggest that smaller w yields spans closer to 5 cm, indicating better alignment. When $w = 1$, where placement fully depends on the vision-estimated centroid, none of the three trials succeeded. A likely cause is observation noise: the measured (x, y) positions of the first two blocks can each shift by millimeters to centimeters, and these errors compound as blocks stack higher. As a result, the computed centroid may deviate from the true tower centroid, causing the third block to be placed off-center and leading to collapse. This also

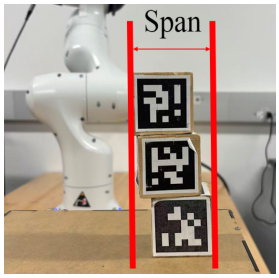

Figure 14: Distance span for blocks

Table 2: Evaluation of weighting factor w.

w	Distance span (cm)
0.00	5.6
0.25	6.1
0.50	6.7
0.75	6.5
1.00	–

suggests that if the system relies heavily on pure vision feedback, accumulated measurement error will grow with tower height and degrade centroid estimation. Therefore, our final system uses hard-coded (x, y) placement ($\omega = 1$) based on the tower base as a fixed reference.

3.2.3 Camera Filtering Window Size

Our framework relies heavily on camera readings, which directly affect overall performance. As described in Section 2.2, we estimate block poses using an averaging window with outlier rejection. To obtain more accurate pose estimates on the real robot, we tune the averaging-window sizes of 10, 50, 100, and 150 using a fixed static-block arrangement (Fig. 15). For each setting, we recorded the total time to obtain four block-pose estimates and the final poses reported in the terminal output.

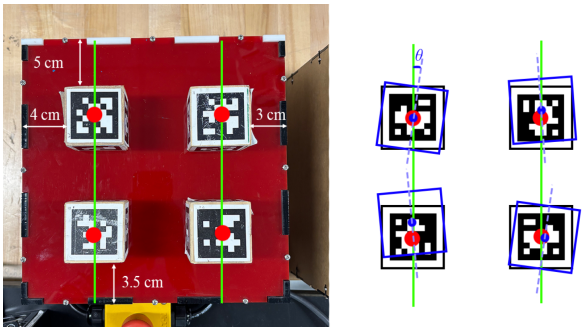

Figure 15: The ground-truth block poses. The blue blocks illustrate example poses readings. For each block, we compute the position error and the orientation (angular) error θ with respect to the ground truth.

To evaluate pose accuracy, we convert each estimated 4×4 transform into block horizontal axes as described in Section 2.3.1. For each block i, we define a joint error as the sum of its position deviation (in cm) and angular deviation (in degrees), $e_i = \|\Delta \mathbf{p}_i\|_{\text{cm}} + \theta_{i\text{deg}}$, and aggregate the overall error across the four blocks as $E_{\text{total}} = \sum_{i=1}^{4} e_i$.

Table 3: Performance comparison across different window sizes for block detection.

Window Size	**Pos. Error (cm)**	**Ang. Error (deg)**	**Joint Error**	**Detection Time (s)**
10	19.80	34.63	54.43	1.36
50	2.36	9.74	12.10	1.71
100	1.81	5.93	7.74	3.37
150	0.67	7.795	8.46	4.01

From the data, a larger window size generally reduces the joint error as more data is averaged and

evaluated. However, this trend is not guaranteed. For example, although 150 observations should theoretically yield a more accurate estimate than 100, our measured result at 150 was worse. This is likely because we did not run enough trials for each window size on hardware, so a single test can be affected by unexpected noise. Nevertheless, the overall trend still suggests that larger window sizes provide higher accuracy: compared to using 10 observations, using 100 observations reduced the joint error by approximately 86%.

One key trade-off is accuracy versus time: a larger window increases observation time and thus slows down the overall system. Moreover, since the same observation module is also used for dynamic blocks, overly long observation windows can become counterproductive when the block moves significantly during sensing: the measurements may no longer capture the initial pose reliably. Considering these factors, we ultimately chose a window size of 100 as a balance between acceptable sensing time and pose-estimation accuracy.

4 Lessons we learned

- The most important lesson we learned is that AprilTag pose readings is highly unreliable on hardware. Although our framework is appealing in simulator, it performed poorly on the physical robot largely because we relied heavily on real-time observation while AprilTag measurements exhibit bias and occasional missed detections. In the first round of the final competition, when placing the third block, the robot pressed down on the tower because it detected only one stacked block instead of two, causing the semi-hardcoded placement height to be underestimated by roughly one block height. This failure was strongly tied to the side-view observation configuration, where the camera could face oblique auditorium lighting that degraded detection. We therefore conclude that for a simple and repetitive task, a complex re-observation mechanism may offer limited benefit while increasing code complexity and reducing real-world robustness.

  A more robust sensing pipeline could mitigate this "missed block" issue. Our current implementation observes once from a single configuration and treats the result as final, making failures brittle. A straightforward improvement is to observe from multiple configurations and fuse the results; our initial code visited four viewpoints so a block missed in one view could still be detected in another. However, we did not adopt this in the final system due to the key trade-off in time: multi-view observation consumes substantial time on sensing rather than execution within the 4-minute limit.

  In addition, we could experiment with more advanced filter mechanisms than our simple mean-based method. Kalman filters, EMA, and other filters are reported to be effective in similar scenarios. It would also be informative to take a large set of data comparing the sensed position and orientation with the exact and characterize the noise systematically. Knowledge of the nature of the noise will allows us to design better filters or counter-noise methods. But over-all, for robust real-world application, the entire perception pipe-line needs re-design. Both the perception hardware and the underlying image processing math should be a studied carefully if similar grasping tasks were to be successfully realized in homes or industry.

- Although our robot implements a complete FSM that provides flexibility and adaptability, this capability was not particularly critical for this competition. The task is primarily to stack blocks, rather than to demonstrate how adaptive the system can be. To some extent, a complex FSM can even reduce stacking efficiency: because it depends on the environment state, it spends more time and steps on observing and updating the world state instead of executing grasps and placements. In hindsight, prioritizing contingency handling (e.g., what to do if the tower collapses) may have been less effective than first prioritizing straightforward strategies to build the tower higher. Moreover, implementing a complex system inevitably consumed hardware testing time, leaving insufficient time for thorough on-robot validation.

A Hyperparameters Used in Final Submission

The code submitted on Gradescope is for **hardware** only (with calibrated camera offsets). To make it work in simulator, please remove all camera offsets in function `get_block_pose_in_FOV` and also change `GOAL_Z_SURFACE_dynamic` to 0.200.

Table 4: Hyperparameter values for final submission.

Category	**Parameter**	**Value**
Coordinate Frames	y_{offset} (robot base offset from global)	0.990 m
	(x_t, y_t) (turntable center in robot frame)	$(0, \mp 0.990)$ m
Perception	N_{window} (static blocks)	100
	N_{window} (dynamic blocks)	2
	τ_{outlier} (Frobenius threshold)	0.01
	N_{scan} (scan configurations)	1
Grasping	w_{block} (block width)	0.05 m
	δz_{pre} (pre-grasp offset)	0.065 m
	δz_{grasp} (grasp descent)	0.065 m
	w_{open} (gripper open width)	0.085 m
	w_{close} (gripper close width)	0.03 m
	f_{grip} (gripper force limit)	55 N
Goal Region	x_{goal} (goal x-position)	0.625 m
	y_{goal} (goal y-offset)	± 0.225 m
	z_{surface} (platform height)	0.200 m
Impedance	T_{descent} (descent duration)	1.5 s
	N_{steps} (control steps)	150
Tower	$H_{\max}$ (max tower height, static)	4 blocks
	$H_{\max}$ (max tower height, dynamic)	5 blocks
	$r_{\text{collision}}$ (base separation)	0.12 m
	r_{cluster} (clustering radius)	0.02 m
	δ_{tower} (tower base offset)	0.13 m
	δ_{side} (lateral approach offset)	0.05 m
	α (position blending weight)	0.75
Dynamic	T_{window} (velocity estimation)	4.0 s
	$\Delta t_{\text{predict}}$ (prediction horizon)	20.0 s
	γ_{wait} (pre-descent wait ratio)	0.6

B Camera Offset for Hardware

Table 5: Empirical compensation offsets added to the translation of the detected T_b^c for each robot

Robot	T_b^c x offset (m)	T_b^c y offset (m)	T_b^c z offset (m)
Red 1	-0.008	0.000	0.0125
Red 2	-0.020	-0.015	0.0075
Blue 1	-0.020	0.005	0.0300
Blue 2	-0.015	0.005	0.0300

C Smoothing Spline Score

names	J
"trigonometric"	0.24387
"cubic polynomial"	0.32818
"septic polynomial"	0.35304
"exponential"	0.36782
"quintic polynomial"	0.3696

Figure 16: Smoothness score of different trajectory functions (direct output from MATLAB). Lower score indicates smoother motion.

N	file	J_int_zddot2
10	"descent_placement_10steps"	0.0062277
30	"descent_placement_30steps"	0.011386
50	"descent_placement_50steps"	0.021788
70	"descent_placement_70steps"	0.051908
100	"descent_placement_100steps"	0.074952
150	"descent_placement_150steps"	0.27922
200	"descent_placement_200steps"	0.28998
500	"descent_placement_500steps"	4.8386

Figure 17: Smoothness score for different numbers of waypoints N (cubic polynomial trajectory).

D Instantaneous Speed at the Last Descent Step

N	TerminalSpeed
10	-0.0016967
30	-0.00051787
50	-6.6667e-05
70	-0.0010181
100	-0.00025613
150	0.0001
200	-0.00018709
500	-3.8294e-05

Figure 18: Robot speed in the last descent step for a cubic polynomial with various numbers of waypoints.

TrajectoryFunctionX	NumWaypointsN	TerminalSpeed
"cubic polynomial"	150	-0.00040909
"quintic polynomial"	150	0
"septic polynomial"	150	0
"exponential"	150	-0.003
"trigonometric"	150	-0.0004375

Figure 19: Robot speed in the last descent step for different trajectory functions.