MEAM 520, Introduction to Robotics, Fall 2020
Final Project: Block Stacking Tournament
Due: Monday, December 14th, 11:59pm

Team 1: Ajay Anand, Hongyou Lin, Griffon McMahon, John (JT) Thompson

May 18, 2022

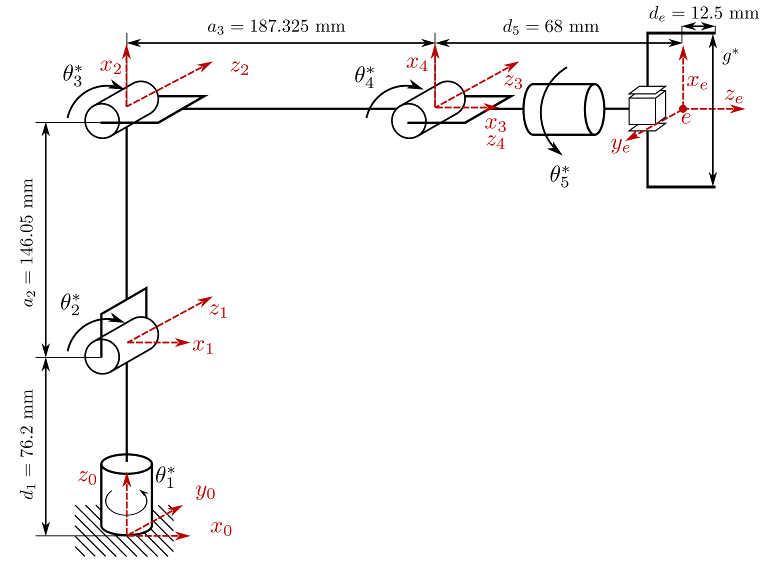


Figure 1: The symbolic representation of the Lynx robot used in the final project.

1 Introduction

In this lab, we will utilize methods taught over the course of MEAM520 to compete in a block stacking competition in teams of four students against other members of the class. After some iteration, we decided to implement a "slap and stack" strategy focused on removing the contested dynamic blocks from the field of play and stacking the static blocks as high as possible. This strategy's strengths lay in the fact that they force a test of how robust an opponent's code is. This strategy performed well in the round-robin, with our team going undefeated. Unfortunately, in the final bracket, the weaknesses of our robot's hard-coded drops led to us losing in the first round. These hard-coded values were largely implemented due to our possession of a Linux machine capable of running the simulation at a 0.99 real time factor, allowing for fast hard-coding. Even so, we are largely content with the performance of our planner.

In order to explain our planner, we have arranged in the following manner. After beginning with the basic ground rules and scoring in section 2, we note the three different strategies we explored in section 3. Then, the actual implementation of our methods is laid out in section 4: general methods in section 4.1, slapping technique in section 4.2, and block stacking in section 4.3. Then, the results of our tests (both in matches and on our personal machines) can be found in section 5; this includes information regarding our choice of final strategy. The actual competition results are included in section 6. Finally, concluding analysis is given in section 7.

2 Ground Rules and Scoring

The competition between two robot arms (red and blue) takes place in a field (shown in Figure 2 with the following specifications according to the final instruction document. Each robot's base (frame 0 in Figure 1) is 200 mm from the world's x- and y-axes and located on the xy-plane. The red arm's base frame is parallel to the world frame, and the blue robot is rotated by 180° about the world's z-axis. A rotating turntable is placed in the center of the world, with its base at the origin. It has a radius and height of 100 mm and 50 mm, respectively. Other environment parameters can be loaded from a given file, `mapfile.txt`. This environment, including the static block and goal platforms, will remain constant throughout trials, but the location of blocks will be randomized between matches. Finally, robots have only 60 seconds of simulation time to act.

The goal of the match is to move scoreable objects ("blocks") onto the robot's goal platform (the green platform in Figure 2). Only the blocks' positions at the end of the match will determine how many points are awarded. A block scores points if it is supported by a team's goal platform. These points are calculated by

$$\texttt{Points} = \texttt{Value} \times (1 + \texttt{Altitude}/10 + \texttt{SideBonus}), \tag{1}$$

where `Altitude` is the distance from the center of the object to the goal platform's top surface (in millimeters), and `SideBonus` is 1 if the block's white side points upward, 0 otherwise. Blocks are 20 mm cubes, coming in two varieties. Purple blocks are randomly placed on the central turntable and reachable by both robots. They have a `Value` of 3. Static blocks are red or blue and randomly placed on static platforms. Each robot has a symmetric pair of platforms within its (and only its) workspace. These have a `Value` of 1. The white side of a block corresponds to the $+z$ axis of that object's coordinate frame. Shown in Figure 3 are three examples of possible final configurations of scoreable objects after a competitive match. The resulting score is displayed for each example.

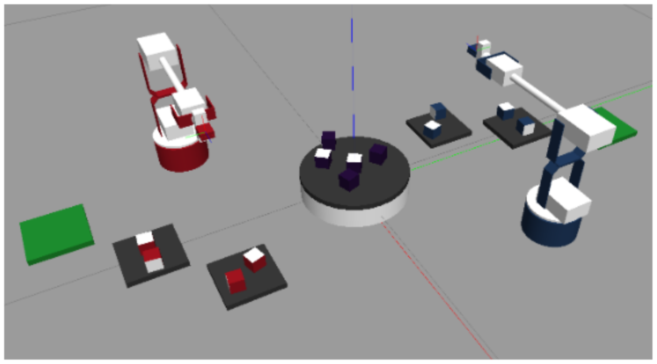


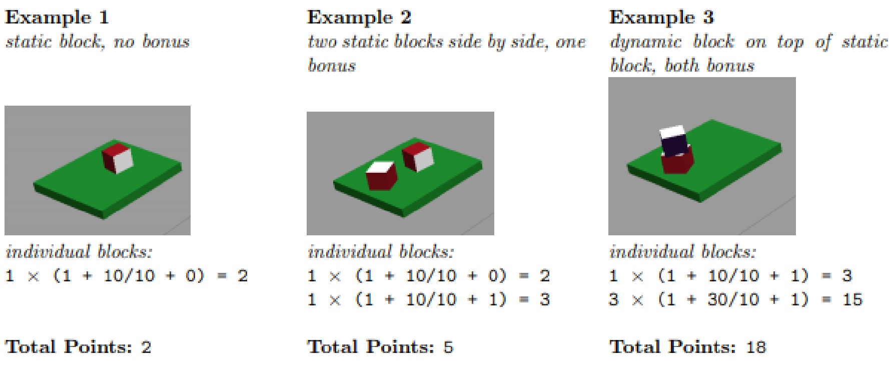
Figure 2: The field of play, as shown in the final instructions document.

3 Strategies Explored

Given the ground rules and scoring parameters explained above, we explored the following strategies which we thought would be successful in the competitive environment. The fundamental idea of our strategy was primarily based on speed and defensive capability. Implementing a planning algorithm to plan the path of the gripper would be computationally expensive and slow the robot down. For the static blocks, since their position is constant, we could simply apply inverse kinematics to calculate a suitable manipulator position to pick the static blocks up.

3.1 Hungry Hungry Hippos

In order to maximize points, we wanted to explore a strategy that successfully places all dynamic blocks on our platform. Because tracking and intercepting a dynamic object in the work space is difficult and computationally expensive, we attempted to simplify our strategy in grabbing dynamic blocks. In the "Hungry Hungry Hippos" (HHH) strategy, we move the manipulator to a suitable grabbing position and move forward to scoop dynamic blocks up as they pass on their rotation around the turntable. The least computationally expensive way to do this



Example 1
static block, no bonus

individual blocks:
1 × (1 + 10/10 + 0) = 2

Total Points: 2

Example 2
two static blocks side by side, one bonus

individual blocks:
1 × (1 + 10/10 + 0) = 2
1 × (1 + 10/10 + 1) = 3

Total Points: 5

Example 3
dynamic block on top of static block, both bonus

individual blocks:
1 × (1 + 10/10 + 1) = 3
3 × (1 + 30/10 + 1) = 15

Total Points: 18

Figure 3: A few examples of points scores resulting from block orientations. Of note is the high score from a dynamic block at altitude (Example 3). These images are taken from the final project handout.

was simply to determine if the center of the dynamic blocks passed close to a line connecting the base of the robot to the origin of the world frame, that is if the ratio of the x and y coordinates of the block was unity.

3.2 Slap and Stack

After implementation and testing of the HHH strategy, we found two flaws in it that would detract from our competitive edge in the tournament:

a) Based on the randomized block placement and the time of the start, the strategy would take longer than 60 seconds to collect all the dynamic blocks.

b) The hippos strategy while computationally simple was slow and hence allowed other teams to also be able to collect dynamic blocks. This would lead to there being a much greater chance of the opponent team winning.

To give us a greater chance of winning a head to head match up, we decided to apply a strategy that was more defensive in nature. The slap and stack strategy was based on the premise of removing the dynamic blocks completely from play during the first few seconds of the competition. The robot would sweep its arm parallel to the surface of the turntable at a very small distance from it back and forth until all the dynamic blocks were removed from play. Following this, the time remaining would be used to stack the static blocks as high as possible. We also implemented a strategy to stack the blocks with the white side facing up for the bonus (if feasible).

3.3 Defense Wins Championships (Final Strategy)

After extensive testing and tuning, we decided on our final strategy for the block stacking competition. In the end, we decided to implement a combination of the HHH strategy and the "slap and stack" strategy, shown as a finite-state machine (FSM) in Figure 4. If executed properly in competition, this strategy all but guarantees victory. Upon start of the match, the Lynx simultaneously attempts to grab one of the dynamic blocks on the turntable while also slapping the rest of the dynamic blocks off the table before our competitor is able to get any. We then stack all of the static blocks with white side facing up (if easy) in a vertical stack and finally place the initial dynamic block on the top of the vertical stack to maximize points according to (1). The basic strategy is outlined below and further explained in section 4.

a) Upon start gun, quickly slide the end-effector with gripper open across the turntable towards the center.

b) Close the gripper and sweep the end-effector from right to left on the turntable to knock dynamic blocks off of the table.

c) Check whether a dynamic block is currently in the grasp of gripper and if so, place the dynamic block on the left corner of the goal platform.

d) Check whether any dynamic blocks remain on the turntable and slap back and forth across the table until all dynamic blocks are gone.

e) After all dynamic blocks are off the turntable, stack the static blocks with white face up in a vertical stack in the right corner of the goal platform. Stack the static blocks closest to the Lynx robot first in order to avoid knocking blocks off the platform.

f) After the vertical stack of static blocks is made, stack the dynamic block on the goal platform on top of the vertical stack. This gives a large `Altitude` bonus to the dynamic block's high `Value` multiplier, maximizing points according to (1).

If executed properly, we grab one dynamic block and slap the rest off the turntable preventing our competitor from getting any. We then maximize side and altitude bonuses to all but guarantee victory.

4 Methods

The actual methods presented here are not the most technical or adaptive, but the team aimed at keeping the planner simple and robust. Overall, the Lynx executes a program as structured in the FSM in Figure 4. As the general strategy is explained in section 3, only the FSM will be described in detail here. The robot moves directly from "START" to "Slap," where the Lynx executes its first slap. Then, if it managed to grab a block in this first lunge, it stacks it in the state "Stack Dyn." If there are still blocks left on the turntable, it returns to slapping ("Slap") until they are all gone. These states are explained in section 4.2. Otherwise, it begins stacking the static blocks in the state "Stack Stat." Once the static blocks have been stacked, it either stacks the initial dynamic block (in the state "Final Stack") or moves to the ending node ("End"). These final states are explained in section 4.3.

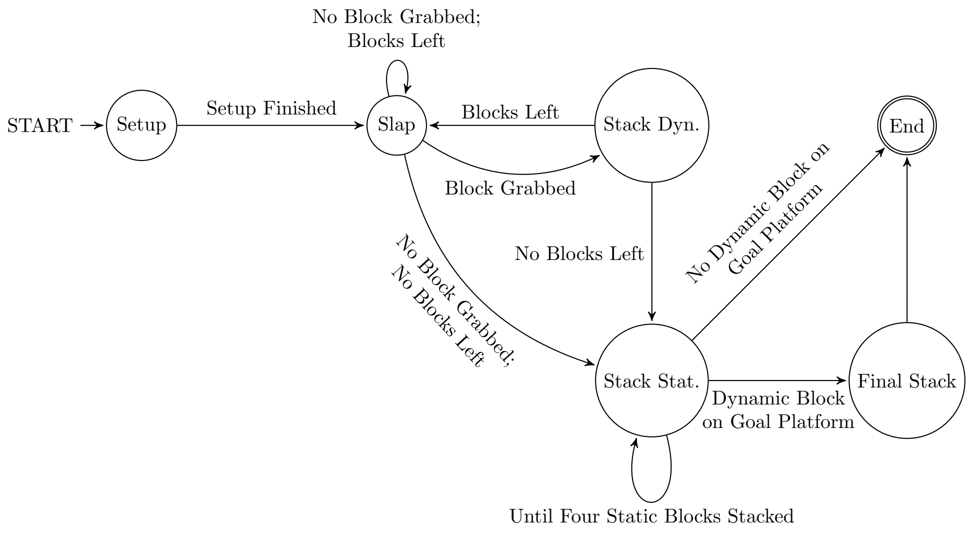


Figure 4: Finite-state machine showing the robot's overall strategy. For clarity, "Blocks Left" always refers to dynamic blocks, as the other team is unlikely to affect the position of our own static blocks.

4.1 General Methods

Some techniques implemented in this project are universal between the different stages of the robot, so they will be included here. Many methods developed in previous labs will not be covered extensively, but they are worth mentioning. For example, Forward and Inverse Kinematics (FK and IK, respectively), are covered in Labs 1 and 2. We will note that FK is used to convert a robot's configuration $q = [\theta_1^*, \theta_2^*, \theta_3^*, \theta_4^*, \theta_5^*, g^*]$ (as defined in Figure 1) into joint positions (represented as homogeneous transformation matrices) with respect to the robot's base frame 0. This can be represented as $T_n^0 = \mathrm{FK}\, q$, where n is the joint whose position is desired. Likewise, IK is used to convert a goal position and orientation for the end-effector (again, represented as a transformation matrix) into a configuration for the robot. This can be represented as $q = \mathrm{IK}\, T_e^0$. Note that, in general, a superscript here corresponds to the frame with respect to which a variable is measured, not an exponent.

Obviously, because the world frame is not the same as the robot's base frame, conversion is necessary; most built-in functions given by the course return positions with respect to the world frame, centered on the turn table. Therefore, we defined two homogeneous transformation matrices

$$H^0_{w,\text{red}} = \begin{bmatrix} 1 & 0 & 0 & 200 \\ 0 & 1 & 0 & 200 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{and} \qquad H^0_{w,\text{blue}} = \begin{bmatrix} -1 & 0 & 0 & 200 \\ 0 & -1 & 0 & 200 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

to convert any world frame-defined position $o^w = [x, y, z]$ or homogeneous transformation matrix T^w into an equivalent one with respect to the robot's base frame. This is done using the equations $o^0 = H^0_{w,\text{color}} o^w$ or $T^0 = H^0_{w,\text{color}} T^w$, where "color" is either red or blue, depending on the robot's color. These matrices were found by visual observation.

In addition to custom commands, some pre-built commands were given by the course. These can be found in Table 1. These commands, along with the universal `sleep()` command, form the backbone of every robot present

Table 1: Built-in functions for the Lynx robot.

Function	**Inputs**	**Outputs**
`lynx.get_object_state()`	—	`name`: Unique identifier for each block.
		`pose`: Position of each block in world frame coordinates.
		`twist`: Linear and angular velocity of each block.
`lynx.get_state()`	—	`curr_pos`: The current configuration q of the Lynx.
		`curr_vel`: The current velocity of the Lynx (unused).
`lynx.set_state()`	q	Slowly but precisely move the Lynx to configuration q.
`lynx.command()`	q	Quickly move the Lynx to configuration q.

in the competition. For one, `lynx.set_state()` and `lynx.command()` are the main methods used to move the Lynx in space to a desired configuration $q = $ `q`. Overall, `lynx.set_state(q)` appeared more useful for precise commands, whereas `lynx.command(q)` was useful for faster movements that required less precision. Although accuracy is always appreciated, the 60 second time limit required `lynx.command(q)` to be used as a time saving measure. Therefore, time was dedicated to deciding which movements required a low enough precision to utilize `lynx.command(q)` instead of `lynx.set_state(q)`.

At the same time, the team found that even `lynx.command(q)` did not act quickly enough for its purposes (especially those relating to slapping). This is largely due to the required `sleep()` function required after each movement function, which allows the Lynx to wait for a motion to be finished before beginning the next motion. As the `sleep()` command used seconds (measured in real time), a simulation environment similar to the one possessed by the course TAs was required to fine-tune the `sleep()` durations. Although the team did own a Linux laptop that ran ROS with a real time factor of 0.99, it desired a way to determine the absolute shortest time the Lynx had to wait before executing the next motion. The method to accomplish this came from a slight abuse of the `lynx.get_state()` command implemented in a helper function named `quick_move(q)` (Algorithm 1). This function simply gave the Lynx `lynx.command(q)`s until the Lynx stopped moving and while the Lynx was too far from the goal configuration `q`. While this the resultant movement was very imprecise, it was suitable for slapping, which largely depended on speed.

Finally, as our strategy depended on testing how robust an opposing team's robot was, our code made use of `lynx.get_object_state()`—assisted with our helper function `cube_dictionary()`—in order to avoid potential bugs, such as waiting for dynamic blocks to enter the Lynx's workspace while there were no blocks left on the turntable. This was accomplished by counting the number of dynamic blocks laying within a cylinder with a radius equal to that of the turn table and a height of 30 mm. The function `lynx.get_object_state()` was also used to determine whether or not the Lynx managed to grab a dynamic block with its initial lunge, to locate our static blocks at the beginning of the match, and to remember where on the goal platform the initial dynamic block landed for purposes of grabbing and dropping it upon the top of the stack.

Algorithm 1: Pseudocode for the `quick_move()` helper function. This function ignores gripper position as a stopping metric, because the gripper moves quickly enough.

```
input  : A desired configuration $q_f$.
output: Fast and imprecise motion.

curr_pos ← lynx.get_state()
curr_diff ← $\|$curr_pos $- q_f\|$                 /* Current distance to the goal configuration */
new_diff ← $\infty$                     /* Also the current distance, but kept for iteration */
while $|$curr_diff $-$ new_diff$| > 0.07$ or new_diff $> 0.5$ do
    lynx.command($q_f$)
    sleep(0.05)
    Update states and find how far the Lynx moved.
end
```

4.2 Slapping and the Dynamic Block

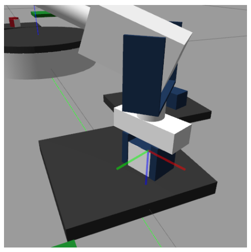
As stated, speed was an requirement for properly executing a slap. If the slap happens too slowly, then an opposing team would be able to grab dynamic blocks before they are all knocked out of play, giving the opposing team an advantage with respect to the number of dynamic blocks available. As noted on Piazza, all robots required a `sleep()` command at the beginning to allow the simulator to set up, even after the starting gun. Though the team initially used `sleep(1)` at the beginning, a scrimmage with Team 2 (and their exceptionally fast dynamic block grabbing) where two dynamic blocks were snatched before our Lynx even moved led us to decrease this wait time. Because attempting to execute commands before setup was completed caused errors, we exploited `lynx.get_state()`'s behavior of returning empty arrays while waiting for setup. Therefore, for the final bracket, the a while loop repeated until the output of `lynx.get_state()` was not empty. This change allowed for immediate execution of the slap, minimizing the time spent in the "Setup" node before moving on to the "Slap" node in Figure 4.

Algorithm 2: Pseudocode for slapping. Every motion except the movement to `qhungry` knocks any blocks in the Lynx's path off the turntable. This represents the code found in the code sections "Slapping" through "Continued Slapping."

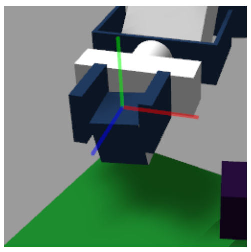
```
q_slapReady ← $[0.3, 0.75, -0.25, -0.67, -\pi/2, -15]$      /* At the table's right side, slap ready */
q_slapped ← $[1.3, 0.80, -0.25, -0.67, -\pi/2, -15]$          /* At the left side, blocks slapped */
if new start then
    qhungry ← $[\pi/4, -0.28, 1.03, -0.54, -\pi/2, 30]$
    quick_move(qhungry) (cf. Algorithm 1)                      /* Ready to lunge at platform */
    qfullopen ← $[\pi/4, 0.72, 0.21, -0.63, -\pi/2, 30]$
    quick_move(qfullopen)                                         /* Lunged into the middle */
    Close the jaws. quick_move(q_slapReady)                                 /* Swept right */
    quick_move(q_slapped)                                                    /* Swept left */
    if block grabbed then
        Exit "Slap," stack the dynamic block.
    end
else
    while blocks on turntable do
        quick_move(q_slapReady)                                             /* Swept right */
        quick_move(q_slapped)                                                /* Swept left */
        Check for more blocks on turntable.
    end
end
```

We also noticed that, during the initial thrust of the snap moving from `qhungry` to `qfullopen` (see Algorithm 2), a block would frequently be within the grabber's fingers. Therefore, the robot will automatically close its hand upon thrusting before executing the first slap. If, by the end of the first slap, there was a block within 20 mm of the end-effector's reference frame, the robot would assume that it had a block in its hand and would drop it off on the goal platform for later. This turns the slap, a mostly defensive tool, into a mechanism for points scoring. This dynamic block is placed to the left of the location that will eventually be the stack of static blocks. After the stack is formed, the dynamic block will be in place to be picked up and added to the top of the stack to maximize points by multiplying its `Value` of 3 with its large `Altitude`.

4.3 Static Block Stacking



(a) A successful vertical grab, with the white side facing the correct orientation.



(b) A horizontal grab, showing the poor block orientation it induces.

Figure 5: The two grabbing orientations of the Lynx

The static block stacking algorithm of our Lynx robot is mainly built on the inverse position kinematics function developed in Lab 3. Once we have the homogeneous matrix of all the reachable static blocks using `lynx.get_object_state()` and (2) for the given robot (red vs. blue), there are three key steps for the block stacking algorithm: 1. Constructing the homogeneous matrix for grabbing based on the desired orientation and the position of the static block for inverse kinematics; 2. Transforming the homogeneous matrix for grabbing into the one for dropping by updating the orientation and positional vector of the desired location; and 3. Computing the intermediate configuration for path planning.

For the first step, two orientations are specified for grabbing as shown in Figure 5a. The vertical grab first attempts to apply a series of rotations shown in Algorithm 3 to the end-effector in order to have the white side up for certain static blocks. As the drop orientation is hard-coded to have the end-effector's y-axis face up (explained later in this section), this algorithm results in the blocks being dropped with their white side up frequently. Although this algorithm ignores "easy" blocks that spawn with their white side facing up, we decided that blocks were more likely to have a sideways-facing white face, so we prioritized those. In case Algorithm 3 does not work, the vertical grab orientation for the end-effector is hard-coded to be $R_e = (\hat{x}_0, -\hat{y}_0, -\hat{z}_0)$ in terms of the base frame unit vectors as a secondary option. For 90% of the time, this orientation works depending on the position of the static block. However, depending on the position of the static block (e.g., when the static block is around the further corner of the lower middle platform), this vertical grab orientation is infeasible. Thus, it is necessary to define another orientation that will always be feasible given the positions of static block, which is shown as the horizontal grab orientation in Figure 5b.

This horizontal orientation defines the intended z-axis to be an unit vector pointing from the base origin to the center of the block on the xy-plane (i.e., the z-component is 0). The intended y-axis is hard-coded to be parallel to the unit z-vector of the base frame, leaving the intended x-axis to be the cross-product of the y- and z-axis. In the program, we prioritize vertical grab because it provides a firm and consistent grasp of the block, which allows for more consistent stacking. In addition, the horizontal grab orientation is also rotated about its x-axis by a small amount (0.1 radians), with the actual position for the homogeneous matrix being slightly higher than the actual block position (this is fine tuned by doing various tests) in order to avoid collision with the platform. This small rotation for better grabbing is later undone for dropping the static blocks.

For the second step, once the robot grabs the block, we transform the homogeneous transformation matrix into one suitable for dropping the block by rotating its current orientation about the base frame's z-axis by the necessary angle of the first joint. In this way, we obtain the desired homogeneous matrix for dropping the block, and the next thing is to apply IK to compute the corresponding configuration. Note that we tuned the height to drop by counting

Algorithm 3: Algorithm for attempting to grab static blocks in such a way that the Lynx can place them with the white side facing up. All unit vectors are with respect to the robot's base frame.

```
input  : pose: the homogeneous transformation matrix corresponding to the block to be grabbed.
output: $R$, the desired orientation of the end-effector as a rotation matrix.

$\hat{z} \leftarrow [0, 0, -1]$                                        /* Try to point the end-effector down */
Rblock $\leftarrow$ pose                   /* Columns are the unit axes of the block's coordinate frame */
upAxis $\leftarrow$ The axis in pose that faces up.
if upAxis is the block's $z$-axis then
    Give up on the white face points.
else
    End-effector's desired $y$-axis ($\hat{y}$) $\leftarrow$ Block's $z$-axis      /* Works because the $z$-axis is sideways */
    End-effector's desired $x$-axis ($\hat{x}$) $\leftarrow$ [Block's $z$-axis] $\times$ $\hat{z}$              /* Cross product to find $\hat{x}$ */
end
$R \leftarrow [\hat{x}, \hat{y}, \hat{z}]$
```

the number of static blocks we have dropped and changed the height of the end-effector accordingly in order to stack better. As a result, the blocks consistently fall 20 mm before hitting a surface, which avoids most bouncing issues. This was implemented instead of placing the blocks on the tower out of fear that imprecise motion or grabbing would cause the Lynx to knock the tower over. In order to avoid the situation of infeasible orientations preventing the dropping of blocks, we also hard-coded a configuration that commanded the robot to drop at the center of the platform, but this rarely occurred in actual tests.

Lastly, once we figured out the desired configurations for grabbing and dropping, the final step was to compute the necessary intermediate configurations for the path of transporting the static blocks. Two of the intermediate configurations were computed by offsetting the goal configurations for grabbing and dropping, and the robot is programmed to raise the end-effector before changing joint 1 for transporting in order to avoid potential collision with the static blocks.

5 Evaluation Testing

In general, our planner has a large dependence on the randomization of the blocks and computation error accumulated in Gazebo, so we decided to run a set of tests in order to correctly evaluate its overall performance at the end. In addition, every time we reach certain stage of the project, we evaluated the performances for these 4 sub-modules of the robot and made necessary updates and improvements before moving to the next stage (outlined below):

5.1 Static Stacking Only

This was the first strategy tested out for the competition; we primarily used the `lynx.set_state(q)` function at first, this led to the manipulator moving very slowly, being only able to stack 3–4 static blocks before the competition time ran out. This was the strategy we testing out in the first scrimmage round. We found that while logically sound, the robot moved too slowly to be able to prove a significant threat in competition rounds.

Following our testing with this strategy we switched to primarily using the `lynx.command(q)` function to pass on our command inputs to the robot, This allowed us to significantly improve our stacking time allowing us to move on the process of picking up and stacking dynamic blocks.

Another issue we noticed during this stage is that the vertical grab orientation will be infeasible for certain static blocks, thus we included a horizontal grab orientation as a backup plan. With various tests and fine tuning of the parameters, we are satisfied (90% of the time) with the performance of the static block grabbing and stacking functionalities and continued to the next stage.

5.2 HHH Method

After seeing our static block stacking strategy work well in our first scrimmage, we then implemented and tested the HHH method. This method consisted of extensive testing and tuning configuration and orientation of the end-effector to maximize the success rate of grabbing a dynamic block (shown below). We randomized the dynamic blocks and tested the HHH method until we successfully grabbed every dynamic block with little to no error.

We found that even with extensive tuning, the method was still relatively slow at grabbing all five dynamic blocks on the table, and the Lynx struggled to grab dynamic blocks on the outer edges of the table. After these tests, it was clear that the HHH method was not optimal as other teams would easily be able to grab dynamic blocks. Additionally, the method was too time consuming and we could eliminate all error in the grabbing of blocks.

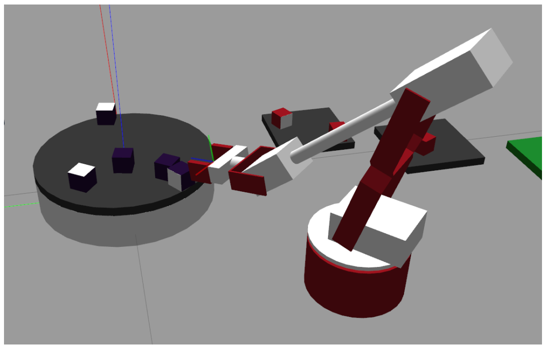


Figure 6: The Lynx ready to either grab a block HHH-style or to thrust into the middle.

5.3 Dynamic Blocks: Slapping-First vs. Grabbing-First

After implementing the HHH strategy, we decided to use a **"scorched earth policy"** to prevent the opposing team from getting any of the dynamic blocks. Part of the strength of this strategy comes from the fact that it acts as a test of how robust the opponents' code was. For example, Team 3 struggled against a slap and stack (despite implementing a slap themselves), because their code never tested if there were any blocks on the central turntable left to grab, so their robot waited in vain for a block to arrive for the entire round. Of particular note was the exciting exhibition match versus the champions of Team 9, in which our planner prevented them from grabbing a single dynamic block. This led to them attempting to grab all their static blocks and crashing when a static block was deemed to be in an infeasible orientation to grab properly. This was to throw off the strategy of teams based entirely on scoring points from dynamic blocks (e.g., Team 9, eventual champions) as well as to give our strategy of stacking blocks white face up an additional advantage (having noticed that most teams did not try to claim the orientation bonus). This strategy, while very effective, working about 90% of the time to grab a dynamic block, does depend on the timing of the starter gun. However, it always prevents the opponent team from getting any dynamic blocks.

The slap alone, while useful, did not completely guarantee success as we expected some of the stronger teams to have implemented a white side orientation strategy as well. This is where the grab and slap strategy that was the cornerstone of our game came into play. Upon firing the start gun, the manipulator would swoop in to grab a dynamic block, followed by quickly slapping the rest of the blocks off the turntable while holding a dynamic block in the gripper. This would give our team the advantage of having an additional block over the opponent to guarantee success.

5.4 Complete Incorporation of Re-stacking the Dynamic Block

All the prior strategies were implemented for the round-robin part of the tournament. All of the teams faced during the round-robin failed to get any dynamic blocks due to our scorched earth policy. Following our qualification into the final round of the tournament, we knew that we needed an additional edge to face and defeat the stronger opponents we would be facing here on in, especially since Team 2 was able to grab a dynamic block even in the face of our strategy. Hence, we decided to implement a higher stacking strategy with the single dynamic block we

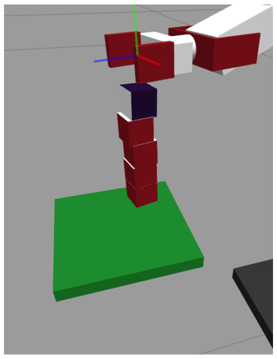


Figure 7: A successful stack, showing the dynamic block on top. All static blocks also have their white sides facing up.

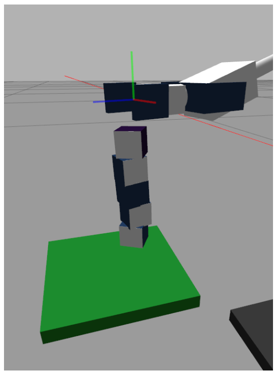


Figure 8: A poorly-stacked tower, with the blocks having different rotations. Their centers of mass are also not aligned, which could lead to the tower tipping over.

attempt to retrieve placed at the top of the stack to maximize our points. The dynamic block retrieved during the grab and slap strategy would be placed on the side of our primary stacking location on the goal platform, followed by stacking the static blocks with the white face oriented up (whenever feasible) followed by moving the dynamic block from the side to the top of the stack. This strategy, while highly effective to maximize our own team's points and denying any dynamic points to the opponent depended more on the randomization of the block map than our previous strategy as the addition of the dynamic block had the potential to destabilize our stack. During testing, we found that in multiple runs with random environments, the stack was destabilized in about 20% of the runs. There were two primary causes behind the stack being destabilized; One being the accumulation of errors in the manipulator control leading to imprecise stacking of the blocks into a tower (e.g., Figure 8), the other being that when the dynamic block was placed too close to the stacked tower, the manipulator would knock the stack over whilst trying to grab the dynamic block for stacking at the top. This proved to be the case in the first round of the finals of the tournament leading to our defeat.

Table 2: Objective data regarding the performance of the planner over 10 trials. The metrics are as follows: "# Stat. Blcks. Lost" is the average number of static blocks knocked off the starting platforms per round, "Stack Size" is the average height of the static block stack in a round, "Dyn. on Top" is the rate at which the dynamic block was successfully stacked on top, "# of White Sides Up" is the average number of white sides facing up and scoring at the end of a round, and "Twr. Pushed" is the number of times the robot pushed the tower while picking up the dynamic block. "1st Try Slap" is the percent success rate of slapping all dynamic blocks off the turntable in one try.

Metric	# Stat. Blcks. Lost	Stack Size	Dyn. on Top	# of White Sides Up	Twr. Pushed	1st Try Slap
Value	0.2	3.4	60%	1.6	20%	80%

An objective test of the planner's capabilities was also run in the form of 10 trials. In these 10 trials, the Lynx never knocked over the static block tower (unlike in the match in bracket) and grabbed a center block with its initial lunge in all 10 trials. The planner only failed to execute completely in one of the 10 trials, when the slap function kept running due to a block close to the Lynx on the turntable was slapped back and forth for 30 seconds instead of being slapped out of play. The less digestible results are shown in Table 2. These results, while not ideal, were deemed acceptable, as the flailing slapping would deny the opponents any dynamic blocks while also stacking a fairly reliable static tower. The mathematics used to ensure that a block could be placed white side up (Algorithm 3) performed rather unsatisfactorily, succeeding in only 1.6 our of 4 possible blocks, though it is suspected that this is due to unfixed bugs in the IK code that would incorrectly deem positions to be infeasible.

6 Live Testing and Results

1. Test 1: December 3rd Scrimmage
   This was mostly uneventful, but we learned that using `lynx.set_state(q)` was not the fastest way to actuate the Lynx.

2. Round-Robin 1: Team 1 (17) vs. Team 2 (2)
   The match up was won as Team 2 experienced problems having their robot start due to versioning problems uploading the final code. Our biggest takeaway from this match was that our imprecise stacking led to the stack falling over.

3. Round-Robin 2: Team 1 (28) vs. Team 5 (12)
   The match-up was handily won. We noticed in this round that when our robot grabbed blocks from the top down, its stacking is much more precise. Additionally, we noticed that our robot successfully grabbed a dynamic block on its initial slap on both of the first two rounds.

4. Round-Robin 3: Team 1 (17) vs. Team 4 (5)
   Although we had another decisive victory, this was the first match that we did not successfully grab a dynamic block on our initial slap.

5. Round-Robin 4: Team 1 (16) vs. Team 3 (0)
   We noted in this round that our slap method was extremely successful as we prevented the opponent from grabbing any dynamic blocks. Our chances of winning going further would be increased with white side up stacking.

6. Friendly 1: Team 1 vs. Team 2
   This friendly against Team 2's updated code exposed a weakness in our strategy. Team 2 was able to grab a dynamic block before we were able to slap them away. We needed to speed up the timing of our slap, as it was delayed due to an initial wait after the start gun.

7. Single Elimination Round 1: Team 1 (13) vs. Team 8 (24)
   The match up was lost due to a fall of the stack. We would have easily won had we not tried to stack the dynamic block on top of the stack of four static blocks, but we wanted to be aggressive to win the championship. More precise placement would have ensured victory.

8. Final Friendly: Team 1 vs. Team 9 Champs
   This friendly match up against the champs proved the strength in our strategy as we handily slapped all dynamic blocks away and created a higher stack than the tournament champions.

7 Conclusion

Although we were satisfied with a quarterfinals finish, there was clearly potential for Team 1 to have taken the title. This was clearly demonstrated in the fact that we took out the championship team in a friendly match following the conclusion of the tournament. Our team's flaws came in the build up of minor errorsinduced by hard-coding values instead of applying more dynamic methods based on the randomization of environments. The most error was accumulated in the grabbing and stacking of blocks. As shown in Figure 5b, blocks grabbed horizontally (the default grasping orientation as described in section 4.3) were not grabbed very solidly. This occurred frequently when blocks were far from the Lynx's base. As such, when they were dropped, they had a tendency to bounce and upset the foundation of the tower. Furthermore, as occurred in quarterfinals, the Lynx did not check the position of blocks surrounding a block it aimed to grab, knocking 0.2 static blocks out of play per round and pushing or knocking the tower it built in 20% of rounds, as noted in Table 2. Despite the prominence of hard-coded values, the robot was quite successful at stacking, with an average stack height of 3.4 blocks and a 60% success rate of placing the dynamic block on top of the tower.

Of the potential improvements to be made, the most clear would be to improve the grabbing orientations for the static blocks. This would avoid a failure in grabbing should the static blocks move. Also, the Lynx should have checked whether the static block was too close before attempting to grab the dynamic block in order to avoid knocking over the tower. On top of grabbing improvements, stacking improvements would have helped the planner immensely. Dropping the blocks from an ever-increasing (constant, assuming the tower is built as expected) height led to poorly-built towers. Implementing placing instead of dropping and re-grabbing the tower as we saw some teams execute would improve our stack consistency dramatically.

The most successful aspect of our planner was the slap, without question. The slap's initial lunge grabbed a block nearly every time, and only prevented the Lynx from executing all commands about 10% of the time (and never in a real match). Therefore, even if blocks close to the edge of the turntable were not reliably grabbed or slapped away, the robot eventually performed as expected. The matches against Teams 3 and 9 both showed that testing how robust an opponent's code was a solid plan, as the slap forced hidden bugs in the opponents' code to surface. While the slap is not a technical approach by any means, it does encourage the fundamentals of good design and made extensive use of forward and inverse kinematics (even if the values present in the code are hard-coded). For these reasons, from the round-robin success to the ruthless efficiency of the slap, Team 1 is happy with its Lynx planner.