Charles Hussey, Hyojung Chun, Christopher Mok, Susan Zhao
Final Project
MEAM 520 Team 07

Block Stacking using the Lynx Arm via a Data-Driven Approach

Table of Contents

Introduction

So far in the class, we have been building the foundation for path planning and exploring options to get a robot from one point from another without contacting obstacles by calculating acceptable trajectories. In the Autonomous Robotic Stone Stacking paper [1], the general problem was to pick up irregularly-shaped objects on the table and stack them vertically. This was achieved with a vision system, a physics engine for pose searching, and a motion planner which was updated after each object placement. This project aims to achieve a similar goal with the resources available. We will be using the Lynx robot to perform actions of locating, picking up, and stacking small blocks. In doing so, we will be exploring several functions which have not been previously done in lab: orienting the gripper and grasping the blocks, accounting for droop and weight of the block and Lynx arm itself, and continuously updating the planner to reflect the changing environment for ensured collision prevention. By developing a solution to this problem, we will gain a stronger understanding of how to finely tune a planner with both velocity, weight, and obstacle considerations.

Methods

Map Construction:

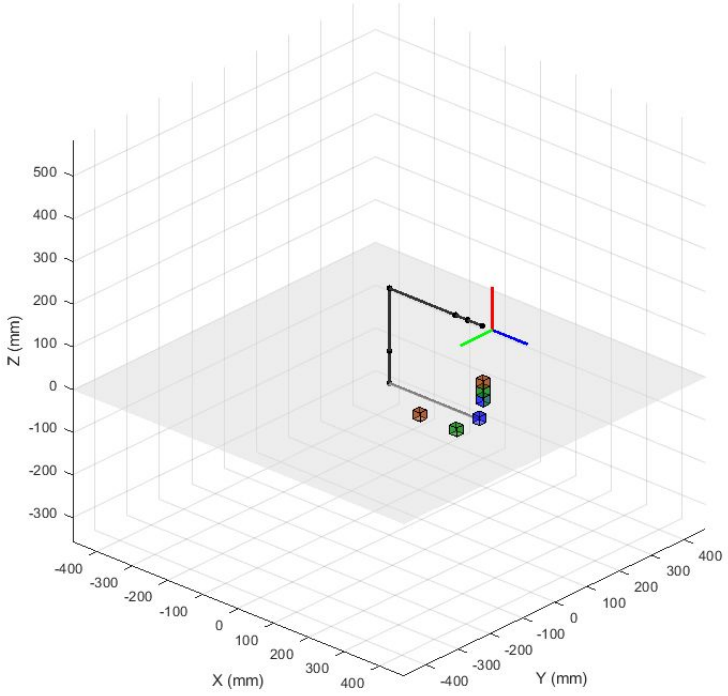


Figure 1: Custom Map

We created a custom map with specified block sizes and location alongside the target stack location and orientation. This was based on the Lab 3 maps, with parameters entered in as:

element x_min y_min z_min x_max y_max z_max

As seen in Fig. 1, the three blocks can be seen in their starting locations as well as the target stack with the colored blocks in the correctly stacked sequence.

To be able to physically simulate the stacking, we created press-fit cubes of length = 0.8in, or 20.32mm. The limit of the Lynx gripper is 30mm. We wanted to maximize the surface area of the blocks for better ease of stacking, but also maintain room for error so the gripper could successfully grip the block. To compensate for both factors we settled on a cube with side length 20.32mm.

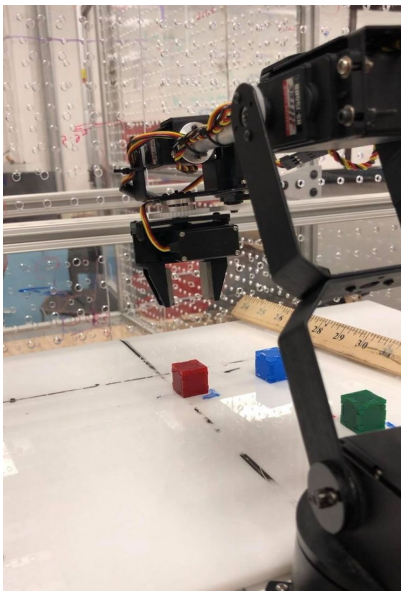


Figure 2 & 3: Blocks used for stacking, Blocks on a platform set at z = 0

Path Planning:

The pseudocode is as follows:

1. Load custom map
2. Generate C-space
3. Execute A* path planning to a reachable configuration within a set tolerance of the block

   Orient the gripper
4. Utilize IK velocity to lower the gripper Close the gripper

   Use IK velocity to raise block above rest location
5. Execute A* path planning to a viable configuration within tolerance of target stack

   Orient the gripper
6. Use IK velocity to lower block onto the stack

   Open gripper to release block

   Use IK velocity to raise end effector vertically up from the stack
7. Update obstacles in space
8. Repeat, starting with A* path planning to reach the next block in the sequence

The basis of our path planning mechanism was the A* search program that we developed for Lab 3, with a few important changes. First, we added the ability to specify different step-sizes for Joints 1 and 2 versus Joints 3 and 4. Simply due to the nature of radial distances, Joints 1 and 2 will likely have a disproportionately large effect on the cartesian location of the end effector. By specifying a smaller step size for Joints 1 and 2, we can achieve a higher degree of accuracy without having to compromise our runtime.

Another important change concerned our c-space generation. As can be seen from our pseudo-code, we used our A* search method to navigate to a point near our desired location. From there, we used Inverse Kinematics to snap more precisely into the position we want to be in before lower to grab the block. An important assumption here is that we could safely move from the end position of our A* search to the position specified by our IK without contacting an obstacle. In order to ensure that we moved a minimal distance during this step, we added a check to our c-space generation algorithm to make sure that the end effector was not only within some tolerance of the correct location, but was also within some tolerance of the correct angle (perpendicular with the ground). Pseudocode for all the functions we used during this assignment can be found in the appendix.

Droop Characterization and Compensation:

A major issue we encountered throughout the semester while running the physical Lynx arm was manipulator pose and end effector position inaccuracy. In nearly all previous lab, there has been significant disparity between the physical manipulator position and that produced in simulation. Theoretically, the onboard PID controller built into the Lynx arm servo motors should correct for any error in position as long as the servos are operating within their torque and angular position limits. However, experience with the actual Lynx arm shows the PID control is simply insufficient, evidenced by occasional collision during the implementation of obstacle avoiding path planners in labs 3 and 5 and our planner's tendency to drag along the surface rather than pick objects off the ground.

In order to achieve end effector accuracy necessary to grasp and manipulate objects, we sought to utilize a data-based approach to characterize the z-position inaccuracy (in all cases observed to be a negative change in z-position) - which we refer to as "droop"- at various positions within the robot's workspace. By gathering a sufficiently representative dataset of droop corresponding to position, we can determine trends in the data that can inform corrections to the position of the robot without requiring servo upgrades or other physical modifications of the Lynx arm itself.

It was noted that the Lynx arm from base to wrist is an RRR serial manipulator in a configuration such that it is essentially a simple RR manipulator that is able to move in planes oriented radially from the Lynx arm's origin within the angular limits of joint 1's servo motor. As such, joint 1's position has no effect on the z-position of the end effector. Accordingly, the end effector droop of the Lynx arm was found at various combinations of radial and z positions.

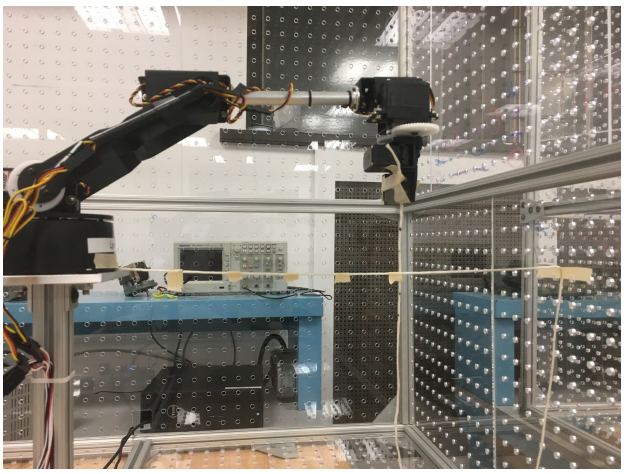


Figure 4: Data Collection Setup

Prior to data collection, end effector positions were predetermined. The collected data points consisted of sets of end effector positions at radial positions of 100, 150, 200, 250, and 300 millimeters and at z-heights of 20, 40, 60, and 80 millimeters. These positions were chosen as they were representative of the positional limits in which the Lynx arm would move and manipulate objects according to the map we constructed. The physical Lynx arm was manipulated until the end effector was positioned precisely at the corresponding z-height and approximately within 3 mm radially of the specified end effector position. The radial position would then be re-measured to ensure characterization accuracy and recorded. The configuration corresponding to this pose was also recorded, and forward kinematics were applied to determine the expected end effector position in simulation. The differences between the simulated and physical end effector position were then calculated for both radial and z axes.

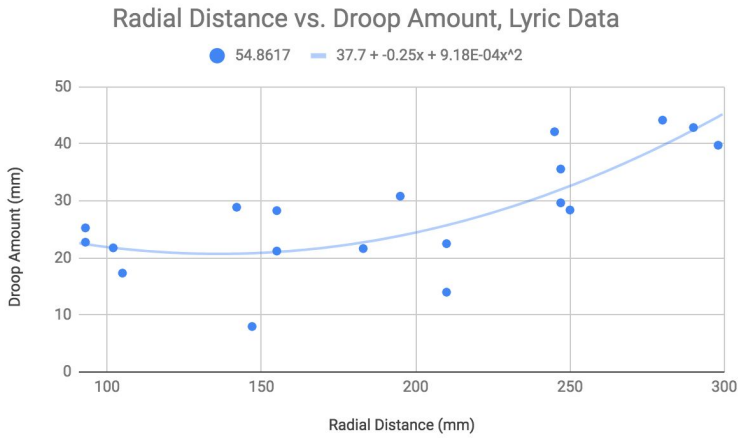


Figure 5: Radial Distance vs. Droop Amount for Lynx Arm Lyric

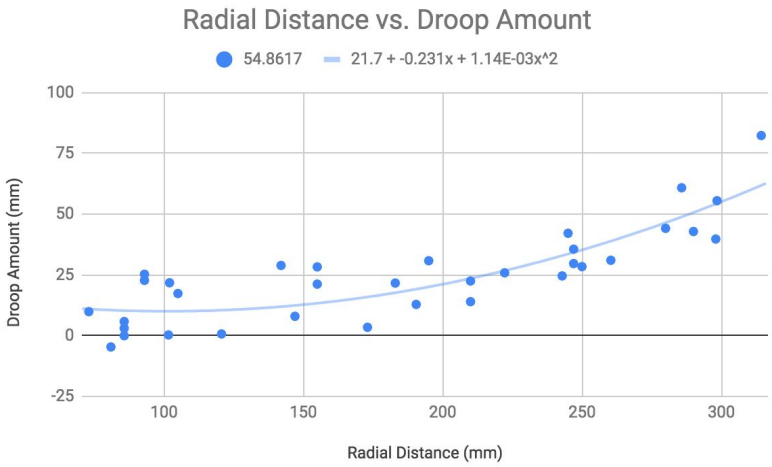


Figure 6: Radial Distance vs. Droop Amount for Lynx Arm (Aggregate Dataset for Legend and Lyric)

When visualized, the collected data exhibits strong correlation between end effector radial position and vertical droop (Fig. 5, 6). Our data set shows parabolic growth in droop with increasing radial distance. We first attempted to use the line of best fit in order to calculate the droop amount at varying radial distance, *r,*, increasing the end effector z-position accordingly, and recalculating the configuration with inverse kinematics. However, this implementation was not an effective means of correction as the inverse kinematic solutions would often alternate sporadically between wrist up and wrist down configurations, leading to erratic end effector behavior and increased inaccuracy.

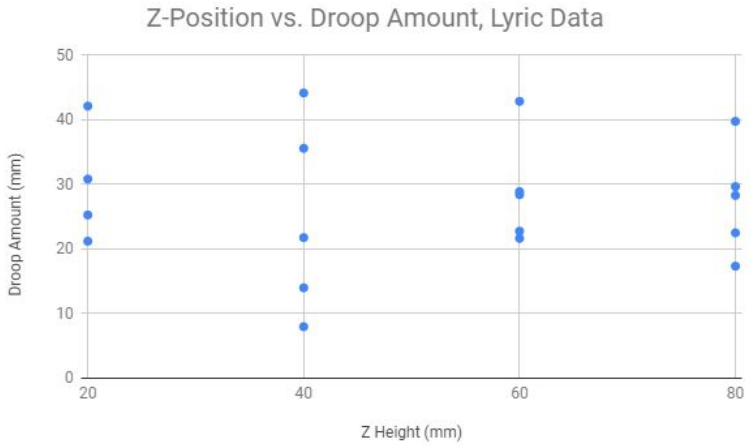


Figure 7: z position vs. Droop Amount

Furthermore, we observed that the amount of droop varied by the height of the end effector (z position) as shown in Fig. 7, but the graph did not elicit a pattern clear enough to draw a line of fit or incorporate its effect directly into adjusting the end effector position.

Based on the shortcomings of the previous attempt at droop characterization and weight compensation, a new method of correction was proposed. In an effort to combine radial and z position into a single metric, we instead opted to compare vertical droop to the center of mass radial position of the Lynx arm. Center of mass radial position was used as increasing this radial position would increase the moment applied on the servos due to the weight of the arm, the suspected cause for droop in the manipulator. Using the same set of data and inverse kinematics, a set of simulated configurations were found corresponding to the physical end effectors positions of the Lynx arm. The center of mass position of these configurations were then calculated using a function we developed which takes into the account relative component weight and configuration, the radial component of which was inspected for correlation. Furthermore, the simulated configurations were compared to the real-world configurations used to reach these locations, and the difference in angular displacements for both joint 2 and 3 (the shoulder and elbow respectively) were compared.

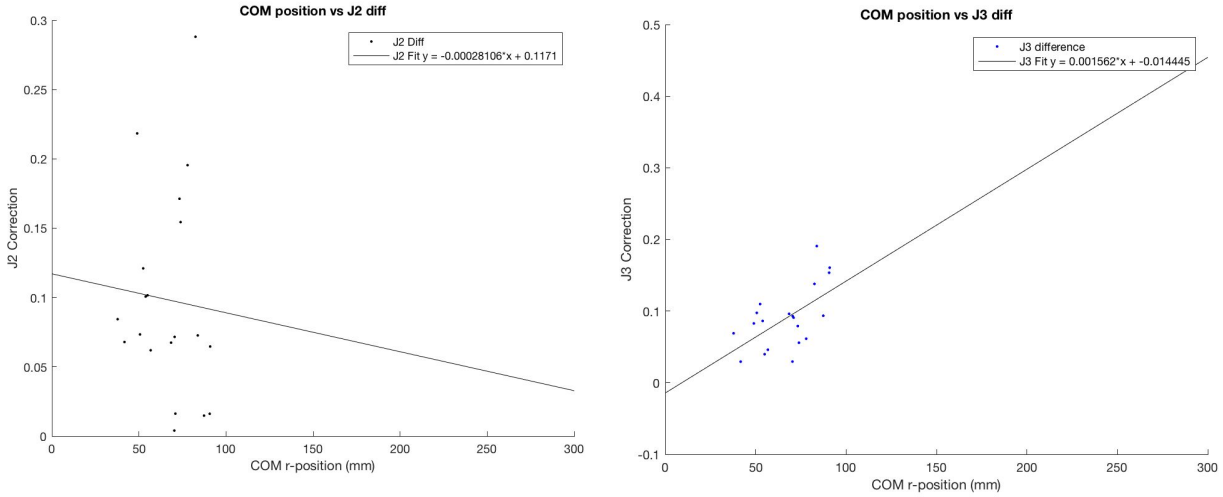


Figure 6 & 7: Center of Mass Radial Positions vs Joint Angle Differences for Joint 2 and 3

Analysis of the data and attempts at curve fitting, revealed that there was no evident correlation between the center of mass radial position and the difference in angular displacement of joint 2 (Fig. 6). However, there was some linear relation between the center of mass position and difference in joint angle for joint 3 (Fig. 7). The relationship was found to be as follows:

$$\Delta q_3(r_c) = 0.001562 * r_c - 0.014445 \quad (1)$$

Where r_c is the radial distance of the center of mass and Δq_3 is the difference in angle of joint 3 between the actual and simulated configurations for a given position.

For our implementation, the center of mass position is calculated for each configuration in the path prescribed by our A* path planner. Using the center of mass at each configuration, a corresponding joint angle 3 correction is calculated and applied to that configuration by the following operation:

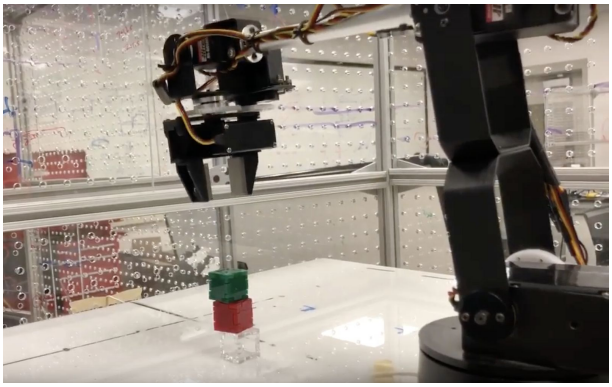
$$q_3 = q_3 - \Delta q_3(r_c) \qquad (2)$$

This corresponding corrected path is then passed into a "readpath" function which controls and actuates the physical Lynx arm through the desired paths from our block stacking function.

Evaluation

Results

Using the path planning algorithm and the equation for droop compensation (2), we were able to stack up to 5 blocks. For the in-class presentation, we showed results for the Lynx arm stacking 3 blocks but after making refinements and adjustments, we were able to stably stack 5 blocks. Specifically, we collected new data to update the variables in (1), and updated our droop compensation function to make adjustments to joint 3, rather than joint 2. We also made adjustments to our path planning function, which allowed it to select more accurate start and end positions.



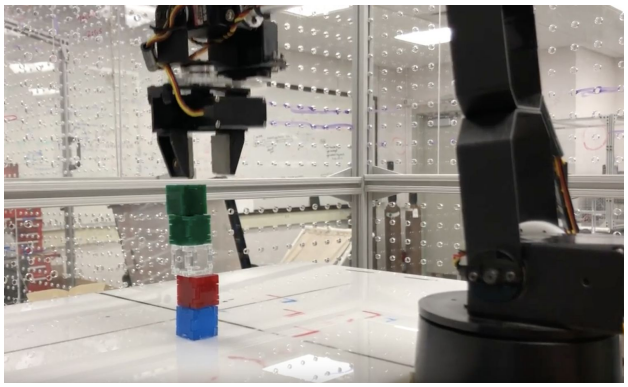


Figure 8 & 9: Stacks of 3 & 5 Blocks

The stacking process can be found in Video 1: Stacking 3 Blocks and Video 2: Stacking 5 Blocks, and Video 3: Simulation for 3 Blocks. Note that Video 1 and Video 3 are in sync and thus when played side by side, you can observe how accurately the Lynx arm is moving.

The path planner still has a number of flaws. Largely due to the fact that the gripper is barely larger than the width of one of our blocks, the Lynx arm has a tendency to brush the blocks as it moves away from the tower. This can lead to poor placement, and can even topple the tower. The Lynx also occasionally hits the blocks as it moves into position to pick them. This is a flaw in our c-space generation, but it is hard to make significant improvements to this without overhauling the method we use to detect collisions with the Lynx arm.

Challenges

Lynx Physical Limitations

We began testing on Robot 1 and 2, Lucky and Legend. In doing so, we quickly realized there were several variables which altered the expected performance of the Lynx arm. There were lost screws which left the servos hinged and out of position. The servos themselves had PID control, but did not match expectations (see *Droop* challenges). Other variables are listed below:

1. Gripper Opening Width

   The gripper width was listed as having a maximum of 30mm, but only reached a fraction of that measurement. On Legend, the gripper opened up to less than 20mm. This was not enough to grasp the solid blocks. As an alternative, we proceeded to use foam blocks (Fig. 10), as they were able to be trimmed down to a usable size. Initially, we considered this a better solution due to the cushioning and absorption of the impact forces upon being stacked on one another. However, issues soon arose, where the foam block would get caught in the gripper after it should have been released (Fig. 11). Ultimately, once we discovered that Lyric could open its gripper to 25mm, we switched back to using solid blocks to eliminate the unpredictable behavior of how the foam sprung back to its original form once it was released.

2. Servo Disconnection

   At times a wire to the servo motor controlling the gripper would disconnect, causing failure for the gripper to open and pick up the block.

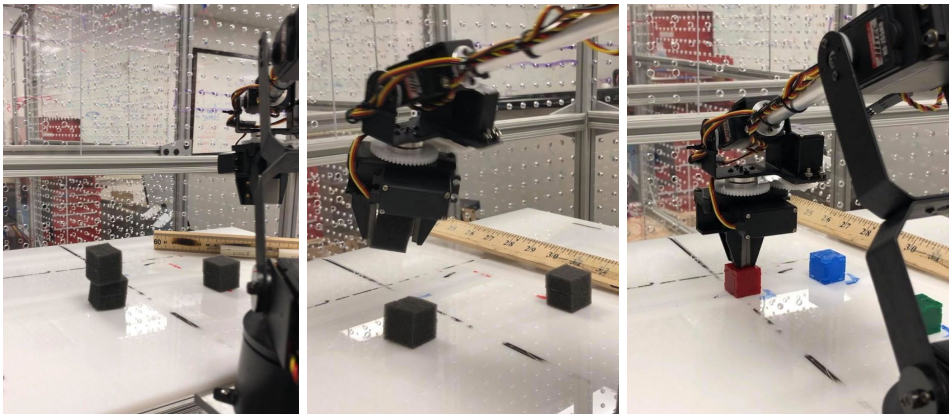


Figure 10, 11 & 12: Setup with foam blocks, error upon release, servo failure

Droop

There were significant complications which arose from the inaccuracy in positioning due to weight and insufficient servo PID control. In events where the Lynx arm brought the second block to the target stack, the bottom of the block would come into contact with the top of the first. This collision was a result of the robot executing a trajectory which, in theory, would not cause any collisions, but due to the system characteristics, caused an error (Fig. 14, 15). In other events, after IK velocity was used to command the robot to move away from the stack, the end effector would come into contact with the stack and alter its position upon moving away towards its next goal.

See [*Droop Characterization and Compensation*] for more information.

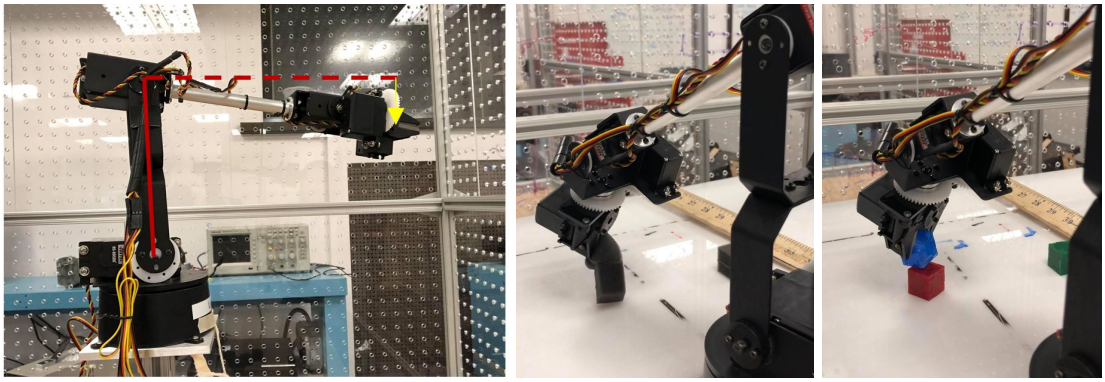


Figure 13, 14 & 15: Arrow depicting deviation from expected position in its zero position, collisions with the stack when attempted to place second block

Precision & Stacking

The precision of the Lynx arm was fairly coincident upon the speed of operation, however not to the point of predictability. We ran tests with different speeds, each with unique results and varying levels of success. With a gripper width barely larger than side length of the block, precision was greatly desired. Slight errors in our placement on the table or deviation from planned trajectories resulted in failure. The gripper would *just* miss the block, cause it to slide out of place and sometimes even cause the servo motor to stall from unexpected behavior (Fig. 16, 17).

There are inherent challenges with stacking objects. With the limitation of the gripper, the blocks were already fairly small. An imprecise placement would cause the stack to become unbalanced (Fig. 18). In our execution, the Lynx arm placed the block by essentially dropping it from a relatively small distance. The block sometimes shifted upon impact, disturbing the balance of the stack.

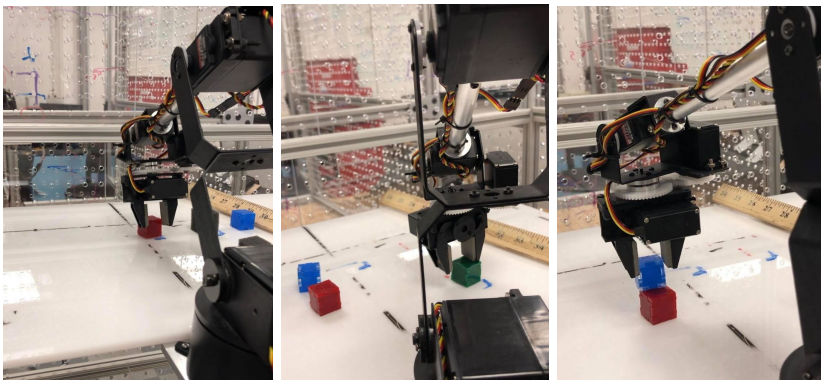


Figure 16, 17 & 18: Gripper misalignment, stacking error upon placement on the edge of the base block

Analysis & Conclusion

Despite a number of initial setbacks, many of which were due to faulty equipment, we were able to achieve significant success with our block-stacking algorithm. Furthermore, we were able to make significant improvements from our progress on demo day, increasing the maximum height of our stack from 3 blocks to 5. We are reasonably confident that we may even be able to go higher, but it may be hard to do so consistently unless we can make improvements to our path planning algorithm.

As described in Precision and Stacking, without any form of droop compensation we observed that the path produced by our block stacking function would often lead to significant error in end effector position and insufficient gripper accuracy to effective grasp and stack objects in our workspace. However, after applying the correction to joint angle 3 according to (1), we observed marked improvement in these aspects. Blocks were grasped at precise heights, without overshoot or premature slamming into the surface of the table. Most importantly, blocks could be reliably at the top of the stack with precision, significantly increasing the probability of placing the block in a balanced arrangement. This increased precision not only allowed us to successfully stack blocks, but also increased the max number of blocks we could stack beyond our initial goal of three.

However, we still concede that the droop compensation is not perfect. For instance, when the Lynx arm attempts to grip blocks that are closer in radial distance, the gripper would grip the top two thirds of the block height, whereas when the arm attempts to grip blocks that are further away, it grips the entire block. Attempting to increase the droop compensation any further results in the end effector gripping the block at an angle. If we were to do further work with this algorithm, we would likely have to balance the effects of increasing joint 3 by adjusting joint 2 as well.

Furthermore, there is room for improvement in detecting obstacles in the map and navigating around them. Although this was outside the scope of our proposal, we attempted stacking after placing large obstacles in our workspace, which the Lynx arm quickly knocked over as it attempted to navigate over it. This is due to the fact that we are using one constant value for the radius of all the joints when calculating the c-space, when the values should vary for each component. This wasn't an issue for a largely obstacle-free environment, but would cause significant issues in a more complex environment.

Overall, we were able to meet the proposed goal of stacking multiple blocks in order using the supplementary droop data we collected.

References

[1] Furrer, F., Wermelinger, M., Yoshida, H., Gramazio, F., Kohler, M., Siegward, R., and Hutter, M. "Autonomous robotic stone stacking with online next best object target pose planning." Proceedings of the IEEE Conference on Robotics and Automation (ICRA). 2017. pp. 2350-2356. doi: 10.1109/ICRA.2017.7989272

Appendix

Pseudocode

exceedLims = checkLims(q):
(checkLims() takes an input set of joint angles, "q", and then checks to see if any of the joints exceeded their angles limits.)
For each angle in q:
    Check that the angle lies within the angle limits of the relevant joint
End for.
If any of the previous checks fail, return true. Otherwise return false.

collisionDetected = checkLynxCollisions(q, obs, bounds):
(checkLynxCollisions takes an input set of joint angles, "q". It also accepts a set of rectangular prisms that represent the obstacles and boundaries of a given map.)
Use the FK function with "q" to determine the start and end point of every link.
For each link:
Use the provided detectCollisions function to determine if the link collides with any of the obstacles.
Check if either the start or end point exceeds the provided boundaries.
End for.
If the lynx fails either of the previous checks for any of the lynx, return true. Otherwise, return false.

[cSpace, q_start, q_end, dTheta12, dTheta34] = generateCSpace(map, dTheta12, dTheta34, radius, start, goal, tol_start, tol_end, tol_ang):
(generateCSpace is the meat of our set of subfunctions, and eats up the most time overall. It takes a map structure, with obstacle and boundary fields. It takes a step-size, dTheta, for joints 1&2 as well as joints 3&4. It takes an approximate radius of the links. It takes a start and goal position, in [x, y, z] coordinates. Lastly, it takes 3 different tolerances for the distance to the start and end point, as well as the angle from perpendicular.)
Take the provided radius and add it to each dimension of the obstacles provided by our map input.
Create a 4D matrix of zeros that represents our c-space (a zero represents an invalid config).
Iterate through every possible combination of values for the four joints:
    Use our checkLynxCollisions function to check for collisions.
If no collision is detected, place a one at the index of our c-space that represents this combination of joint values.
Also, check to see how far the end effector is from desired start/end positions/angles. If it's nearer than the previous nearest config, update to reflect the new nearest config.

End Iteration
If the iteration ends, and our minimum configurations aren't within the specified tolerances, then we halve our step-size and run the function again.

pathOut = aStarSearch(cSpace, dTheta, q_start, q_goal)
(This function takes as an input our c-space. It takes our step-size, dTheta. It also takes the valid configurations, q_start and q_goal, that represent the start and end positions, respectively.
We create 4 arrays for our A search method:*
exploredNodes: keeps track of which nodes (in which order) we have explored
parentNodes: keeps track of which node immediately follows every explored node in the path back to the start node.
neighborNodes: keeps track of the set of all neighbors of exploredNodes that are valid configurations.
neighborNodeVals: keeps track of the value of each neighbor node, using the A function we defined earlier.*
)
Add the start node to exploredNodes and add a 0 as the first entry in parentNodes.
While exploredNodes does not contain q_goal:
Let node "n" be the node most recently added to exploredNodes.
Use our getPathLength function to find the length ("g") of shortest path from the start node to node "n".
Call our addNewNeighbors function, using "n" and "g" as inputs.
Take the resulting list of neighbors and append it to the existing list of neighbors.
Take the resulting list of neighbor-values and it to the existing list of neighbor values.
Sort our list of neighbor-values in descending order.
Whatever switches we made to our list of neighbor values, apply the same switches to our list of neighbors.
Remove the last element in both the neighbor and neighbor-values lists, and add this node to the end of our exploredNodes list.
Add to the end of parentNodes the index of the parent of this new node.
If the set of neighbors is ever empty at this point, and the end node hasn't been added to exploredNodes, that means that there is no path from our start node to our end node. In this case, we just return an empty set.
End While.
Create our path variable and add our goal node to it.
Let j be the index of our goal node's parent (i.e. j = parentNodes(end)).
While j does not equal zero:
Add the node at exploredNodes(j) to the end of our path.
Redefine: j = parentNodes(j).
End While.
Add the start node to the end of our path.
Flip our path from left to right, and return the result.

[newNeighbors, newNeighborVals, cSpace] = addNewNeighbors(parent, index, cSpace, start_index, end_index, dTheta, g)

(This function takes a number of inputs. The first input is the index of our input node in the exploredNodes list. The second input is the indices of the node whose neighbors we are trying to find. The third input is our c-space. Next, the program takes the indices that correspond to our chosen start and end configurations. The function also takes our step-size, dTheta. Lastly, our functions take an input "g", which corresponds to the minimum path length from the start node, to the node whose neighbors we are currently considering.)

Let us call the node whose neighbors we are discovering, node "n".

g = g + 1

For each of n's 8 neighbors:

Call our checkLims function to make sure that the new angle configurations don't exceed the limits.

If the angle config is within the limits, check our c-space to see if the configuration is valid.

If it is, add the config to our newNeighbors list.

Find the normalized distance, "f", between the new neighbor and our end node.

The value of our new neighbor = f + g.

Add this value to newNeighborVals.

Add a zero to our c-space at the index of our new neighbor.

End If.

End if.

Add an extra row to the bottom of our newNeighbors list that just contains the value of our input "parent".

g = getPathLength(parentNodes)

(getPathLength takes a single input: the list of parent nodes. It finds the length of the path found by traversing backwards from the current last node, all the way to the start node)

Let j be the value of the last index in parentNodes (i.e. j = parentNodes(end)).

g = 0.

While j does not equal zero

Redefine: j = parentNodes(j)

g = g + 1

End While.

path = findpath(map, start, end, dTheta12, dTheta34)

(findpath doesn't actually do much beside make calls to other functions)

Initialize various constants (i.e. radius, tolerances)

Call the generateCSpace function to get our c-space, as well as our start and end configs.

Call the aStarSearch function using the outputs of the previous function.

Return the output of our A*function.

path = stackBlocks(map, cube_map, tower_loc)

(stackBlocks runs the main loop of our block stacking method. After initializing a variety of things, it loops through each block, navigating from block to tower, then vice-versa. Note that the function doesn't actually perform these functions with the physical servo, but rather saves each step to an aggregate path, which it then outputs)

For each block
    Use A* to navigate to a spot above the block
    Use IK_velocity to lower the gripper onto the block, grab it, and raise back up
    Use A* to navigate to a spot above the tower
    Use IK_velocity to to lower the gripper, drop the block, and raise the gripper again
Endfor

CoM = findCoM(q, jointNum)
(finds the radius of the approximate center of mass of the lynx arm, after the joint specified by joint num)
Use calculateFK_sol to find the joint positions
Use estimates of link weights to find the location of the center

path = fixDroop(path)
(fixDroop takes in a path, and applies an adjustment factor to joint 3, determined experimentally)
Call findCoM for each step.
Adjust each joint 3 value accordingly.

lynxServoPaused(q, q_last)
(Tasks in the next configuration and the current configuration. Makes calls to lynxServo)
Find the difference between the new and old configuration
Compare the difference to the max angular velocities for each joint to find how long we need to pause for.
Pause, then call lynxServo()

path = readpath(filename)
(takes in the filename of a file to which a properly formatted path has been saved. Adjusts the path and plots it.)
Scan in the path file into a useable format
Call fixDroop to adjust the path
Start the lynx
Make calls to lynxServoPaused to actuate the lynx

Test
(Used to conduct tests. Specifies a set of parameters, saves the output to a useable file)
Load in the obstacle map and the block map
Call the stackBlocks function, and save the output to a text file