ME134 Final Report

Rajeev Datta
Shivansh Gupta
Lucas Lanzendorf
Christopher Zhou

Task

Our core functionality is stacking square quadros on a 3-D printed baseplate. The robot should track placed quadros in the workspace and continually stack them on available stacks of quadros on the baseplate.

During execution, we anticipated a set of reasonable actions people would take when interacting with the robotic system. Our system handles if someone moves the location of a block during execution. Specifically, if a block is moved into the workspace, the camera system detects the additional block and updates the available blocks for the robot to sample and place from. If a block instead moved into the stacking workspace (the area defined by a clamped baseplate), our system recognizes the placed block as the start of a new stack and adds it to the plausible stacks to sample from and stack onto. Someone can even be particularly mean and remove a block from an existing stack. In such a case, the detector system will update the stack height and continue stacking onto the newly altered stack in future time steps. Lastly, using a depth camera, people can stack blocks in workspace and the robot will, if sampling a stack, stack the highest block in the chosen stack.

During the implementation phase of the project, we faced several core challenges associated with our task. First and foremost, the necessary positional accuracy to pick up quadros in the correct position was a persistent issue. This issue caused significant shifts in system design on both the hardware and software level. Moreover, building on the previous issue, the act of stacking quadros required additional considerations outside of improving positional accuracy.

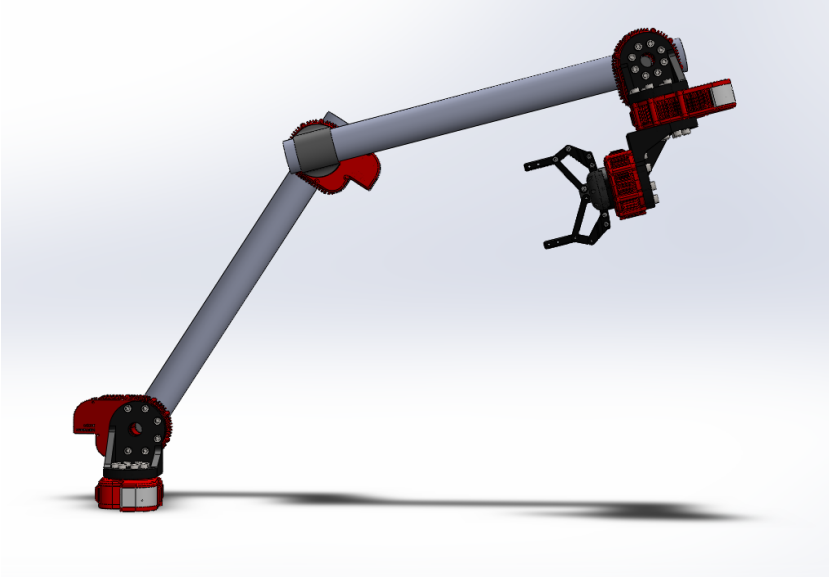
Hardware

Given our challenge of designing a robot to manipulate LEGO Quadros, Some design considerations when building this robot include the motor selection, arm design, a means of attaching motors to the arms and to each other, and gripper design.

Given a series of HEBI X5 and X8 motors to choose from ranging in continuous torque ratings, some napkin calculations were performed to decide which motors to place at each joint. Given our longest moment arm of approximately three feet and the total weight of the motors taken into consideration, we approximated 15 Nm of torque at the shoulder joint, 7Nm at the elbow, and 3Nm at the wrist. X5s are universally lighter than X8s which meant it was critical to constrain all X8s to the smallest moment arm possible. The first three motors were therefore X8s with the weakest being placed at the base where it would be largely unaffected by gravity on the arm. The motor forming the shoulder joint would experience the greatest magnitude of torque therefore the most powerful X8 with a rating of 16 Nm of continuous torque was placed at the shoulder. The elbow joint used the final X8, capable of providing 9Nm of continuous torque

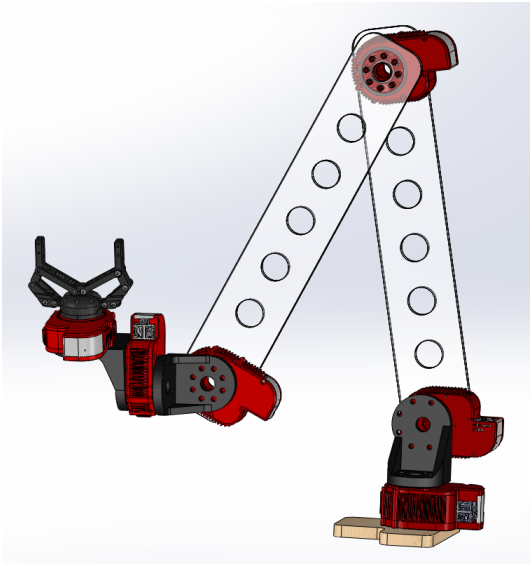
which surpasses the expected 7Nm it will experience normally. The X5s experience a similar selection process with the weaker motors being closest to the gripper and strongest being attached to the second arm. The chain of three X5 motors connected by angle brackets at the end of the entire arm were all able to operate well within their operating range given the small moment each would be experiencing.

The initial design for the arms featured one-inch inner diameter aluminum pipes connected to the HEBI motors using 3D-printed brackets secured into the pipes using two transverse screws on each bracket. The geometry of the pipe and its material provided great rigidity to the arm which is desirable in properly transferring simulated movements into the real world with the greatest accuracy possible.



Initial design of the robotic arm with two 18 inch pipes as arms

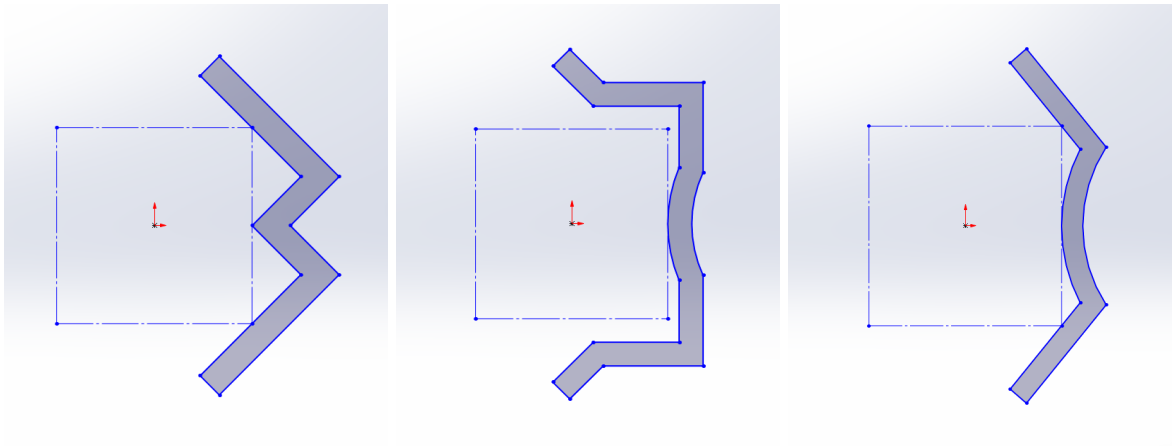
Unfortunately we were unable to get the accuracy we desired from the pipes which we thought to be a problem with our URDF. It took much trial and error but in the end we opted to switch to laser-cut acrylic for the ease of debugging URDF problems to fully narrow down what was causing the error in our real world accuracy. These acrylic arms were 18 inches as the pipes were to maintain the same workspace as before. Two identical quarter-inch arms were pieced together to create each section for increased rigidity and the acrylic was screwed directly into the motors to simplify translating between each link in the URDF. While this is less rigid than the aluminum pipes, we were successful in resolving the errors using these acrylic arms and chose to keep them through to the end.



Finalized CAD of the robot as of demonstration day
(modified grippers excluded)

Many iterations of the gripper have existed with both large design overhauls and simple modifications. The gripper sold by HEBI was the initial design used in early tests however the original goal was to design a soft gripper cast out of molded silicone which would inflate and deflate using a pump controlled through an arduino and stepper motor mechanism. This ultimately failed due to difficulties utilizing silicone as a medium for building with ruptures and leaking of air being far too prevalent to replace the standard parallel grippers. Reflecting on the design also raises some concerns on whether it would have been an optimal design to begin with. Due to the compliant nature of a material like silicone, the block is unlikely to have been consistently oriented relative to the grippers which would make stacking very difficult. Seeking rigidity for repeatability we turned to 3D-printed solutions, initially replicating the parallel grippers provided with an additional one centimeter in width when fully opened to accommodate the size of the blocks. This was successful in gripping the blocks with the addition of high friction adhesive like athletic tape but would often grab blocks off-center and subsequently miss in placing the blocks because the robot assumes the block would be perfectly centered within the grippers.

The next step in the design process was to find a method of centering the blocks within the grippers using the geometry of the grippers to move the part as it closes around.



Initial sketches for self-centering grabber chronologically from left to right, rightmost design was used in final design.

Tests of the final design were printed in PLA through the Caltech Tech Lab and were very successful in centering blocks within the gripper. By wrapping in athletic tape like the previous design, friction was sufficient to hold the blocks securely.

Kinematics and Planning

Task coordinates: x, y, z and e_x, e_y, and e_z with the origin defined at the center of the axis of the base motor. We utilized velocity inverse kinematics to convert task space movements into corresponding joint movements.

The angular velocity Jacobian was also employed to control the orientation of the end effector. Specifically, we kept the the orientation of the gripper normal to the table surface. When the robot reached for blocks to pick them up, it would adjust the angle of the gripper so that it was normal to the unit direction vector of the block.

The inverse kinematics were computed by first determining the joint space velocities. This was achieved by inverting the 6x6 Jacobian matrix and multiplying it by the combined velocity/angular velocity vector along with the error i.e.,

$$\dot{q} = J^{-1}(\dot{x} + \lambda \cdot err)$$

Quintic splines were applied to produce smooth movements without sharp discontinuities in velocity or acceleration. The spline we implemented was applied to both tip and joint movements. We also used joint movements to move the robot's joints into adjacent solutions spaces (something about 8 multiplicities and the corresponding solution spaces). For instance, if we grab a block from the side, we utilize joint movements to change the wrist motor joint positions to retain the elbow up configuration while moving the end effector to place the block away from the base. We also splined joint movements to ensure smooth movement to home

positions no matter the current position. One example of this is when the robot starts up, it first lifts itself vertically, from its current position, to avoid colliding with any objects that are laid out on the table.

To generate continuous commands, we used information from the image processing node to obtain position information of blocks that could be grabbed and blocks that could be stacked upon. By combining this together in the main node, we were able to produce trajectories on the fly that would position the arm for grabbing and stacking.

To facilitate localization accuracy, we also created a gravity model on both shoulder joints and the first joint of the end effector. By sending commands of nan to both position and velocity, we were able to test how well the arm could "float" by just utilizing the gravity model. The following torques were commanded:

$$\tau_3 = -0.6\sin(\theta_1 - \theta_2 - \theta_3) - 1.2\cos(\theta_1 - \theta_2 - \theta_3)$$
$$\tau_2 = -6\cos(\theta_1 - \theta_2) + \tau_3$$
$$\tau_1 = -\tau_2\cos(\theta_1 - \theta_2) + 9\cos(\theta_1)$$

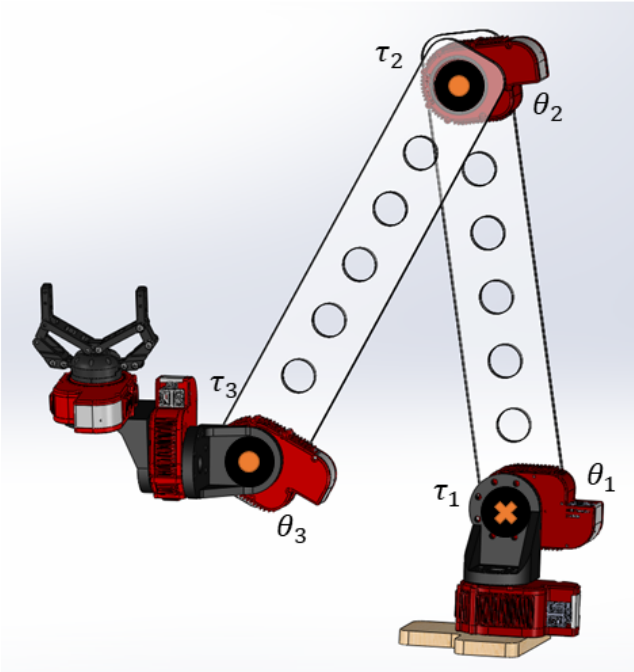


Diagram showing corresponding theta angles and torque vectors for the shoulder motors, as well as the first joint of the end effector. Here, τ_1 *is pointing into the page, while both* τ_2 *and* τ_3 *are point out.*

Camera Detection

The goal of our detector was to find pieces on the table, and separate them into available pieces that hadn't been stacked and pieces that were already stacked on the baseplate. To do this, we attached ArUco markers on the front of each face (the face with studs). We wanted to read these markers and get the X,Y,Z coordinates for these markers with respect to the baseplate such that we could send these coordinates in a message that our main demo node could read these and command our arm to go to these positions to pick up/stack the blocks.

For our camera, we used an Intel RealSense D435 depth camera. This camera is mounted approximately 79.2 cm above our table, and we mounted it by clamping a baseplate to the vent overhead and then by attaching the camera stand to the overhead vent.

Our camera detector itself is a separate node in our software architecture. We made use of both the RGB and the depth images. The depth images came in handy when trying to determine the height of the topmost block. This was essential information when the robot created taller and taller towers.

To obtain calibration information we subscribed to the /camera/color/camera_info topic to obtain the distortion coefficients and the camera's intrinsic matrix K. After obtaining this, we were able to compute the undistorted pixel coordinates from the camera, by calling the undistortPoints method in OpenCV and passing in this distortion coefficients and camera matrix.

To ensure we could convert the camera's pixel coordinates to the coordinates in our world frame (essentially measured from the center of the base motor clamped to the baseplate), we taped 4 ArUco markers to the corners of our table, and we supplied the world coordinates of each of these ArUco markers in our script. We also modified the script such that everytime our detector ran its process function, it would supply new pixel coordinates for the ArUco markers. We would then generate a new matrix M that mapped these pixel coordinates into the world coordinates everytime the process function ran. The benefit of updating the pixel coordinates continuously was that even if our table was to move accidentally during the demo or overnight, the arm would still go to the correctly supplied world frame location. This is because our baseplate was anchored to the table, and so our measured coordinates were still the same even if our table moved. To get the actual world coordinates from the pixel coordinates, we made use of the getPerspectiveTransform() function in OpenCV and supplied the matrix M that we generated in the process function.

We used the RGB camera to get the X and Y coordinates with respect to the base motor attached to the baseplate, which was then clamped to the table. We used the depth camera to get the Z coordinate from the table. Originally we obtained this by subscribing to the topic with the depth

image that was aligned to the color image. This was the distance d from the camera to the block, and we subtracted this distance from 79.2 (the distance from the camera to the table) to get the height at which the block was at.

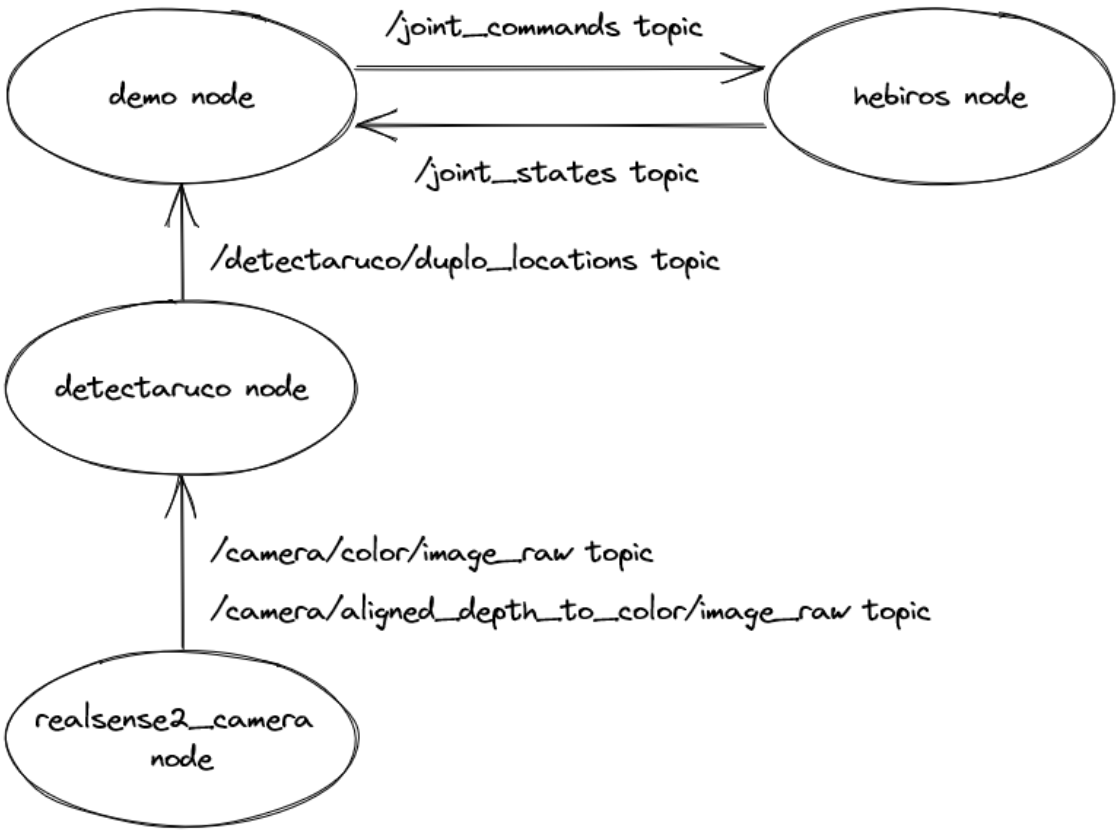
We noticed later on that the perspective transform was not as accurate when the object was near the periphery of the camera's field of view. So, to fix this projection error in the X and Y coordinates, we used the following formula. Given a point $p = (x, y)$ that we wanted to correct, and the point directly underneath the camera $p_0 = (x_0, y_0)$, our new coordinates for (x', y') would be:

$$\frac{(0.792 - z)}{0.792} (p - p_0) + p$$

Another unique thing that we did was that we used a time synchronizer in ROS to combine both the RGB and depth images into one so that we could have access to the RGB image and depth information simultaneously.

Software Architecture

Our software for this project consisted of two nodes. The first node, named demo, controlled robot behaviors and movements, while the second node, named detectaruco, processed live images from the RealSense camera and identified blocks in the workspace. The interactions between the two nodes are visualized below (along with the hebi and camera nodes):

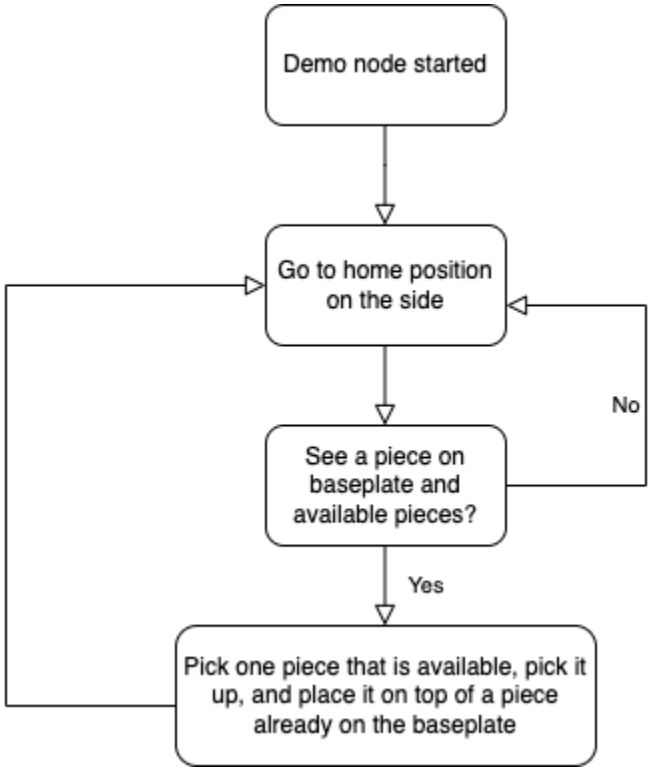


The detector node detectaruco identified blocks by their ArUco marker and converted their pixel coordinates into Cartesian coordinates in the world frame. These world frame coordinates were gathered together, along with the direction vector of each block into a single Float32MultiArray message. The detector node would publish this message to the /detectaruco/duplo_locations topic continuously, as new RGB images and depth maps were received.

In the demo node, which subscribes to the /detectaruco/duplo_locations topic, these messages are processed, and used to generate new trajectories for the arm to grab and stack blocks together. The demo node also communicates with the hebiros node to send new commands to the motors, and receive updates on the joint states.

Behaviors and Failure Recoveries

The primary behaviors for our robot can be summarized by the diagram below:



While this diagram is simple, it has many implications that lends itself to easy recovery and safety mechanisms. Our robot automatically goes to a home position off to the side so that our detector is able to clearly see the table. This allows it to easily find pieces on the baseplate as well as available pieces in the workspace region. Now, if the detector is able to see a piece on the baseplate and any pieces in the available regions then the demo node will be notified. The demo node will then send commands to the motors to pick up one of the available pieces and to place it on top of a random piece in the baseplate region. Once this is done, the robot goes back to the home position off to the side to once again allow the detector to clearly see the table.

This is simple functionality, but some failure mechanisms are inherently handled with this behavior model. In particular because nothing is hard coded, the robot will never start moving erratically towards the Lego baseplate if there is nothing available to be stacked upon. Likewise, if there is no piece available in the delineated workspace region, the arm will simply stay in its home position. Moreover, the robot is smart enough that if a block is added or removed by a person, or if it changes locations while the robot is running, the robot will still be able to stack an available block on top of a block in the baseplate (as long as one exists). We also support the functionality of having multiple pieces or stacks on the baseplate, so the robot can pick up any

available piece and choose any block to stack it onto. Additionally, if the robot fails to pick up or put down the piece, it is up to the user to place the piece back in the available region, otherwise the robot will not see it. Another very useful fail-safe feature of our detector is that even if the table moves slightly, the robot will still be able to grab and place the piece accurately. This was very useful especially in the off chance that people bumped into our table during the demo.

Shortcomings

One flaw in our robotic system is that it can only recognize blocks if they are rightside up. While working through the project, we attempted several times to implement a color detector that could recognize blocks no matter which way they were arranged on the table. However, we found that this detector was not robust enough to always draw accurate enough bounding boxes around the blocks. Moreover, the detector could not always identify every block on the table surface, even if we placed only blocks of the same color.

Because of these issues, we decided to use a ArUco detector system instead, which would identify blocks in the rightside up orientation, as long as it could see its ArUco tag. This adjustment enabled our system to be more robust, since ArUco markers are very distinctive and easy for the camera to detect.

Another observation we made was that when the arm lowers itself down to stack a block, it tends to apply more pressure to one side of the block then the other. This makes the process of stacking more challenging, since ideally, the arm would apply equal pressure to all sides of the block, allowing it to more easily lock in. For this reason, we ended up 3D printing 2x2 quatro blocks, so that there would be greater tolerance, and make it easier for blocks to be stacked.

While testing/running our demo, another issue we noticed was that if there was a stack that was adjacent to another, shorter stack and if the robot tried to place a piece on the shorter stack, the robot would accidentally knock over the taller stack. Something that we would need to work on is intelligent piece placement such that the tip could rotate and avoid collisions between adjacent towers.