America, Davis, and Eloise

ME/EE/CS 134

20 March 2025

Final Report: Shelvin’ Cooper the Robolibrarian

Introduction: Task and Concept

Shelvin’ Cooper the Robolibrarian is a 6-DOF robotic arm that assists with sorting and shelving books. With Shelvin’, users can place books within the workspace (provided they are not overlapping each other) and allow the robot to handle the sorting and placement of these books on the shelf. This is intended to demonstrate technology that can assist librarians by reducing re-shelving time without having to significantly interact with the robot.

Because shelving books is multi-stepped, visually complex, highly coordinated, and innately human, this project was highly complex. Six DOFs are needed to accurately pick up, rotate, and place books; markers and cameras are needed to determine book orientations and shelving availability; and a complex state machine is needed to drive the transitions between each of the steps. These difficulties and our design decisions are described throughout this report.

At a basic level, the robot first uses vision to detect a book on the table. Then, it reorients the book so that the spine is parallel to the edge of the table. Next, the book is moved to the pickup zone, from where it is finally picked up and placed in the next available shelf slot. Each of these steps and the various considerations that must be designed for are also described in the sections below.

Mechanism and Hardware

The design of our robot is a more complex extension of the initial 3 DOF arm designed for the class. We use 6 degrees of freedom including a shoulder horizontal rotation, a shoulder vertical rotation, an elbow, a wrist vertical rotation, a wrist horizontal rotation, and a gripper vertical rotation (in addition to the gripper motor itself). The HEBI X8-9 is used for horizontal shoulder rotation. A stronger motor is used here because it controls the pan motion of the entire arm. Next, a HEBI X8-16 motor is used for the vertical shoulder rotation. This is the strongest motor available for us, and it was necessary for lifting the heavyweight of the 6 DOF arm. An X5-9 motor is used on the elbow as it is again lifting a significant amount of weight, while X5-4 motors are used on all remaining joints because they handle less weight and need more precision and speed. An X5-9 motor is used for the gripper cable actuation. The robot, shelf, table, and book were modeled in Onshape. When the arm stretches to shelve a 1lb-book, the torque exerted about the X8-16 motor was approximated to be about 16 Newton-meters, not including the weight of the links. Since the X8-16 motor can exert a peak torque of 40 Nm, we decided to 3D print and laser cut the parts and iterate upon the design if necessary. The CAD model is linked here:
https://cad.onshape.com/documents/76729ec31bc2a3db5b03c44d/w/8ac03c09c996d3fde7f649f5/e/ce2b411d71e195d5db2d8012?renderMode=0&uiState=67dc832fd70f565d5e910d5d.

We use two links, one of length 415 mm between the shoulder and elbow, and one of length 315 mm between the elbow and wrist. A spine structure is used on each link to provide rigidity while being lightweight. A 3D printed U-Bracket houses the shoulder and gripper motors, while two L-shaped brackets build the wrist and actuator structures. Finally, the gripper consists of two 3D printed pincers that are covered with anti-slip rubber pads to help with picking up the books.

The URDF was constructed via the onshape-to-robot Python package, which seems to require one assembly/part per link. Otherwise, onshape-to-robot will fail to generate the URDF. In addition, the CAD model should be in its zero position before running the converter. Our CAD model fed into the converter is here: https://cad.onshape.com/documents/76729ec31bc2a3db5b03c44d/w/8ac03c09c996d3fde7f649f5/e/33063c214a4dc9d4385974f5?renderMode=0&uiState=67dc8be4d70f565d5e912c2d.

Besides the use of a 6 DOF robot, one of the most unique features of our system is the use of two separate tool tips for the kinematics. The first is the gripper tip, which is used for picking up and shelving the books. The second is the bracket tip, which is at the end of the gripper L-bracket. This tip is used for reorienting and pushing the books into the pickup location.

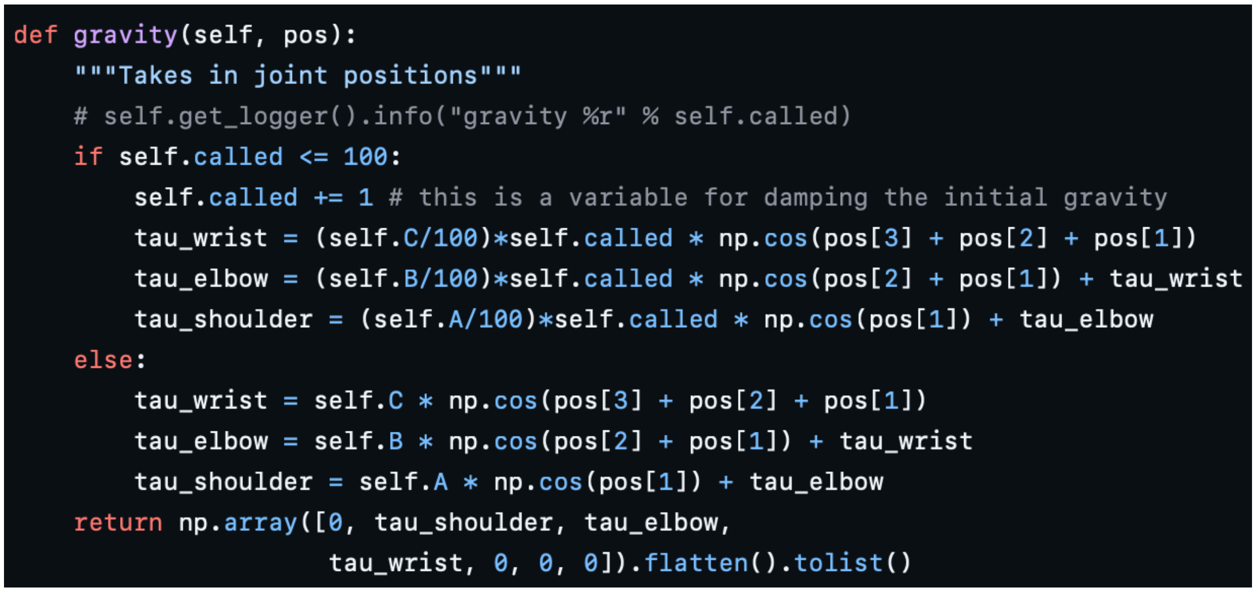
Lastly, we also use several books donated by America and Eloise, and the Caltech Library generously lent us an old table shelf with shelf dividers.

Task Space, Kinematics, Trajectories, Precision, Continuity

Given that shelving books is an innately human task, it involves both translational placement in all x, y, and z directions, along with all orientations (at some point during the algorithm). Thus, our task has 6 degrees of freedom, although we mostly focus on two angles for the book's orientation. This many DOFs required a more careful approach to the Newton Raphson so that reasonable solutions, aka solutions in which the robot didn't collide with itself or the table, were determined. First, we initialized the robot to its home position and stored its joint positions as an initial seed. The Newton Raphson algorithm uses the initial seed to compute how to move the bracket tip to the correct x-coordinate, while maintaining the same y-coordinate. The final joint positions are stored as used as the next seed for computing how to move the bracket tip to the correct y-coordinate. We use the final joint positions as the following seed for computing the joint positions during book pickup. If the robot must move the bracket in the x direction first, then the y direction, the robot will perform these actions. Otherwise, it "hallucinates" this process: Newton Raphson is run, but the joint positions aren't published in a SegmentArray message. In addition, we use the joint positions at the home position as a seed for computing the joint positions during shelving. When shelving manga as opposed to Western literature, the gripper must be rotated -180 degrees instead of 180 degrees. Strangely, Newton Raphson finds unreasonable configurations with the former gripper orientation. Therefore, we compute all the joint positions assuming that the book is Western; if the book is manga, we manually set the gripper's joint position to -180 degrees.

Furthermore, although the orientation of the gripper during reorientation and translation doesn't matter (so long as the gripper doesn't scrape the table), Newton Raphson struggled to find reasonable solutions using the bracket tip chain and a Jacobian for the desired x, y, z coordinates as well as the desired normal vector for the z-axis of the bracket tip. Thus, we specified the desired x, y, z coordinates and rotation matrix for the bracket tip. To ensure Newton Raphson is able to find a solution in 6-DOF task space with only 5-DOF joint space, we almost always limited the bracket tip frame to rotate about the world's z-axis . (The exception is the moment right before the robot shelves the book. The bracket faces slightly upward to prevent the book's foot from colliding with the shelf). The Newton Raphson algorithm uses a weighted inverse to prevent the robot from reaching singularities. As we corrected the URDF file, we wrote pi and pi/2 to several decimal places to ensure Newton Raphson could converge in under 10^-6 error.

The gravity model is a very important module within the trajectory node. Since both the robot arm and the books it carries constitute a much greater weight than we dealt with in the 3 DOF case, we needed a very strong gravity model. The equations for the gravity model are shown below. Note that since the coefficients are quite large, this effort command would introduce a jerky movement when the robot started up. This was improved by slowly ramping up the gravity over each time the module gets called by trajectory. We found that increasing the coefficients 1/100th at a time produced a smoother startup.

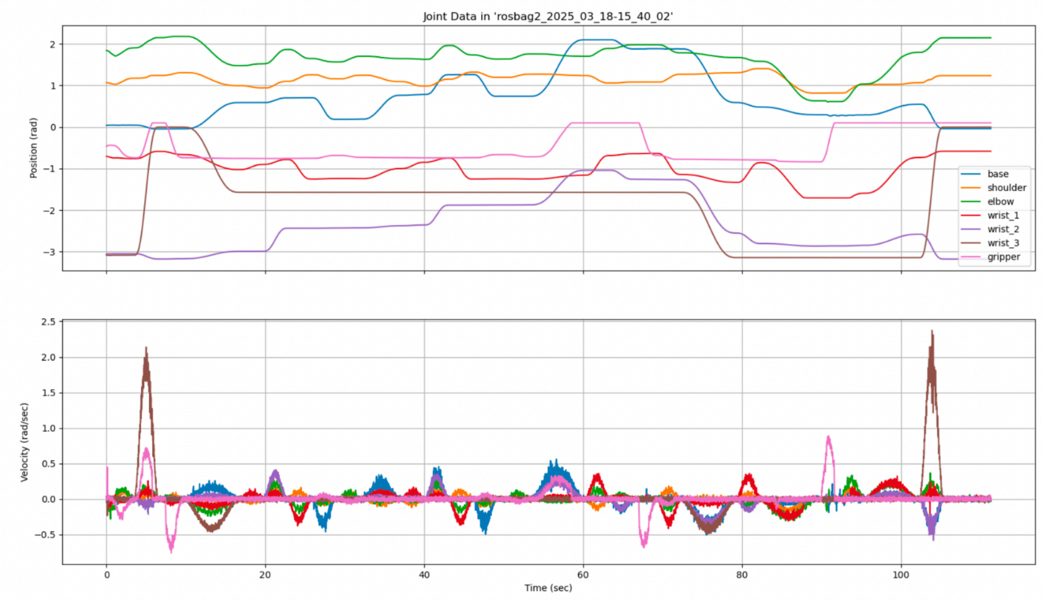


```python
def gravity(self, pos):
    """Takes in joint positions"""
    # self.get_logger().info("gravity %r" % self.called)
    if self.called <= 100:
        self.called += 1 # this is a variable for damping the initial gravity
        tau_wrist = (self.C/100)*self.called * np.cos(pos[3] + pos[2] + pos[1])
        tau_elbow = (self.B/100)*self.called * np.cos(pos[2] + pos[1]) + tau_wrist
        tau_shoulder = (self.A/100)*self.called * np.cos(pos[1]) + tau_elbow
    else:
        tau_wrist = self.C * np.cos(pos[3] + pos[2] + pos[1])
        tau_elbow = self.B * np.cos(pos[2] + pos[1]) + tau_wrist
        tau_shoulder = self.A * np.cos(pos[1]) + tau_elbow
    return np.array([0, tau_shoulder, tau_elbow,
                     tau_wrist, 0, 0, 0]).flatten().tolist()
```

Where A = 30, B = 8, and C = 2.5.

Likewise, moving large weights around at unique orientations meant that the robot would have different errors across the table and across task objects. This encouraged us to produce a large error map which greatly increased the accuracy of our system. The friction between the table and the bracket impacts the positional error of the tip, therefore we always lift the tip, then place it at a point (next to a book). Different orientations of the tool tip at a given point had different translational errors as well, so we calculated the error map for when the bracket tip was in one of three orientations: when the z-axis of the bracket tip faces negative x, positive y, or positive x. Whenever the robot receives a desired position and orientation, the positional error at the desired position is interpolated between all points collected for that orientation, and we adjust the commanded desired position accordingly. This made the pushing motion and picking up motions significantly more consistent. If the bracket tip is commanded to be at a different location, e.g. when it's parallel to the fore edge of a book, we simply tell the robot to interpolate the positional error for all points collected at the positive y orientation. This interpolation process could be more effective if we interpolated the error across the orientations. The force that the book exerts on the robot also impacts positional error of the tip. To facilitate pushing books the proper amount to the pickup zone, we associated a push offset with each book i.e., if a book must move -0.1 meters in the y direction, the robot will be commanded to push the book -(0.1 + the push offset) along the y-axis.

To demonstrate a typical cycle of the robot, we recorded the joint positions and plotted them in the figure below. There are a few interesting results to notice in this plot. First, the base joint remains relatively close to the zero position except for when the book is being translated or needs to be picked up from the side of the table. Second, the positions are very smooth, which is expected given that we used quintic splines to program our trajectories. Third, the highest velocities are recorded by the smallest joint around the wrist. This is desirable because we needed the joints lifting a heavier weight to move slowly so that the robot would not oscillate. However (and fourth), there is still some oscillation that can be seen in the elbow and base around 90 seconds. This is likely where the robot shelved the book before returning home. Since these values were tuned close to the shelf, and since the robot released the weight of the book, there was some extra movement at a high enough speed to cause the robot to oscillate.

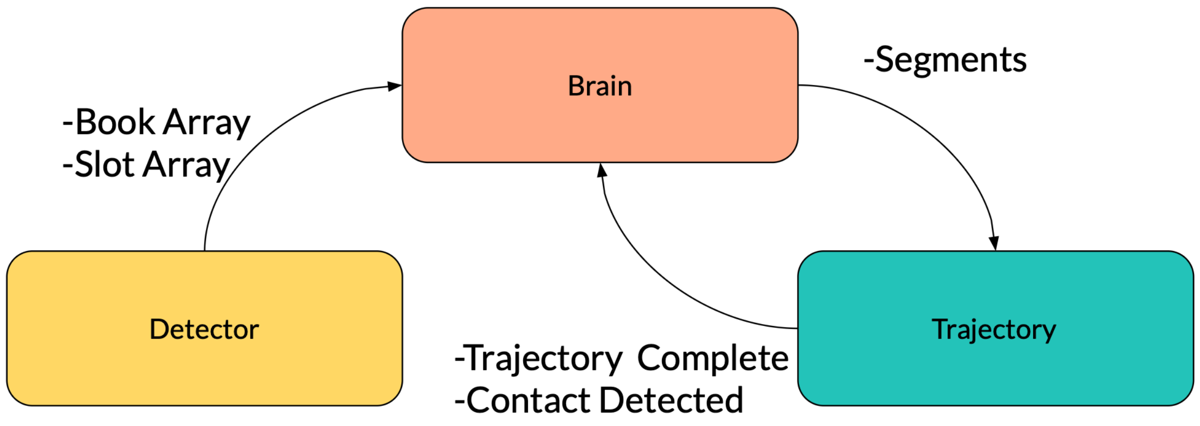


Cameras and Visual Detectors

The system continuously processes images from the USB and RealSense camera. The RealSense RBG images are used to detect all Aruco Markers (table markers and book markers). Using the first image, the table markers are used to compute and store our perspective transform (self.M). This perspective transform is then reused for every book marker detected to calculate its corners (and therefore its location and orientation). The inverse perspective transform (self.Minv) is also calculated at the beginning, once, and is used to annotate each image with the four corners of every book detected. The USB RBG images were used to detect available bookshelf slots. Our original approach was to define rectangular boundaries for each slot, threshold the HSV values of the image to produce a binary image that distinguishes the wooden shelf color from other colors, erode and dilate the binary image, then use the mean pixel value within the slot boundaries to determine whether each bookshelf slot was occupied or available. The night before demos, the USB camera successfully detected the slots, and the detector node published a SlotArray message, containing a list of bools, to the brain node. The brain would use this message to determine the leftmost empty slot and direct the trajectory node to shelve the book in this slot. The next day, however, the USB camera node kept dying shortly after being launched, even though v4ls-ctl said the USB camera was streaming on a certain port. Restarting the NUC would resolve this issue, however, the problem would sooner or later reoccur. Hence, toward the end of demos, we instead started with an empty shelf and instructed the robot to shelve from left to right.

ROS Software Architecture

The following flow chart presents the central nodes of our ROS program as well as the messages sent between them.



First, the detector publishes a book array (containing the four corners of each book, whether or not the book is manga, the book's push offset, and the book's ID) and a slot array (containing indicators for whether one of the five shelving slots is available or full). The raw images with annotations are also published. These are used for debugging with RQT and are not interpreted by any other node.

The brain contains the state machine for the robot, which will be discussed in the next section. In addition to running the state machine, the brain completes all of the most complex calculations for the trajectories. Using the book and environment measurements produced by the detector node, the brain decides whether a book needs to be reoriented, translated to the pickup zone, and placed in the shelf. Each of these activities consists of position and orientation goals that the robot uses, along with its previous joint states, in an iterative Newton Raphson to determine the goal joint states. These desired Cartesian coordinates might include aligning the bracket or the gripper with the edge of the book and rotating proportional to its angle, pushing according to its planar location, or picking up and placing it according to the available shelves. The Newton Raphson works iteratively to confine the solution space to a reasonable configuration, as discussed in the Kinematics section. Once all of these calculations have been completed, the brain sends a segment array to the trajectory node.

The trajectory node handles the physical movement of the robot. Using the segment messages it receives, it calculates the spline the joints will move through. When the trajectory is complete or when the robot notices a collision event, it sends a message back to the brain to indicate that it needs new segments or needs to go home.

There are a couple coding intricacies that were used during coding implementation so that the system would work correctly. First, the trajectory node will only accept one array of segments at a time. For example, it will accept the sequence of picking up and shelving a book, but not in addition to another movement to reorient a different book. This allows us to control the number of times a segment is added (ideally, only once per segment) and it requires the robot to always take notice of changes in its environment before calculating the next move. Second, the trajectory node will only publish a trajectory complete message once. This is done to prevent the brain node from responding to multiple complete messages and sending the same segments multiple times. In addition to these methods, the book messages were only received with a buffer of 1, while most other messages were sent or received with a buffer of 10. This helped in reducing the number of times the robot saw a book and created a full trajectory.

Our code base can be found in the following Github repository: https://github.com/eloisezeng/me134 (in the goals10 branch)

Modeling/Understanding the World

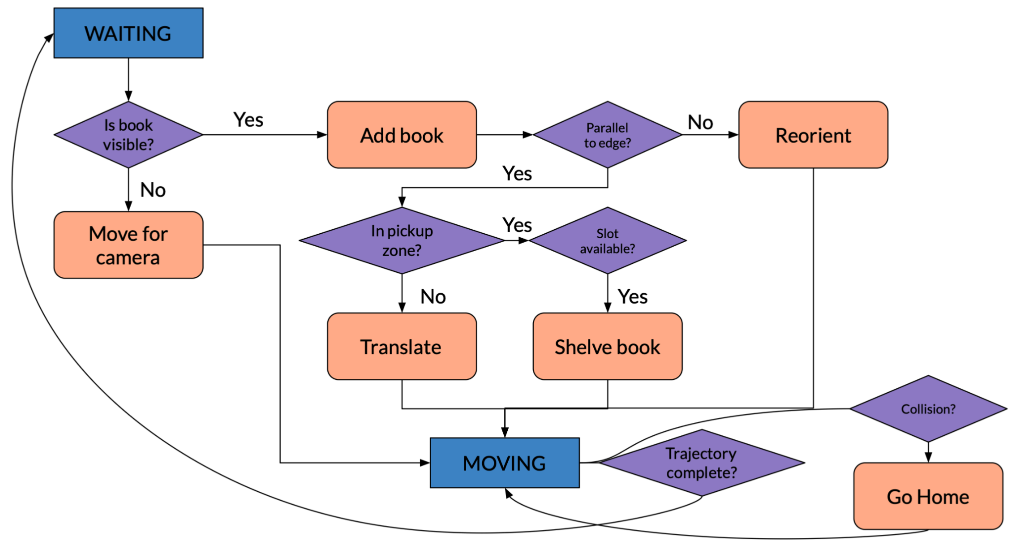
As shown in the node diagram in the previous section, there are several messages and callback functions that are utilized throughout the operation of the robot. These are used by the brain node to move between different states of operation. This state machine consists of Init, Waiting, Moving, and Contact. The Init state is only used at startup to move the robot to its home position. In the Waiting state, the brain accepts new book messages from the detector node and makes the necessary calculations for the trajectories. This state occurs only in between movement actions so that books are not added multiple times or incorrectly (if the robot is blocking the camera). The Moving state locks out the addition of new books, as well as the publishing of new segments to the Trajectory node. This state switches back to Waiting only when the trajectory node signals that it has completed the current segment array. Finally, the Contact state is used to handle collisions or unexpected movements of the robot, and it can interrupt other states to be handled as needed. These states can be seen in the sharp-cornered boxes in the behavior flow chart shown in the next section.

Through this method, we are able to exert a high degree of control over the states of the robot and which information is being handled at a given time.

Behaviors and Failure Recoveries

The standard operation of the robot is described in the flowchart below. Since the system continuously updates the detected objects and slot occupancy status, the robot always reacts based on the current workspace. First, the robot checks if a book is reachable (i.e. if it's within 0.1 m and 0.7 m of the robot base). After a book is reoriented or translated, the robot will check if it can still detect the Aruco marker of that book. If not, the robot moves its base by 30 degrees toward the home position. If the book is visible now, the robot rechecks whether to reorient, translate, or pick up the book. If the book is still not seen, the robot returns to its home position then waits for the detector node to publish a new list of detected books. If a certain action did not accurately place a book where the robot wanted it to be, the robot would retry the action (according to the new environment) until it was satisfied. Lastly, the robot would always manipulate the book closest to one of two pickup zones. This allows us to handle multiple books on the table at once, react to users moving the book between robot movements, and select the proper slot to fill. Using the secondary USB camera over the bookshelf, the robot will also select the available slot or refuse to shelve a book if the shelf is full.

Collisions are also handled as part of the Contact state (not shown below). The brain will command the trajectory node to erase the remainder of the current segment array and replace it with the "go home" sequence. A collision is detected in the trajectory node using feedback from the HEBI motors. The trajectory node checks whether the positional, velocity, or effort error has exceeded a threshold of pi/4, pi/3, or 12pi respectively. This helps filter collisions so that only actual undesired contacts (such as with a human) are responded to in this way, while changes in velocity (such as the robot lightly hitting the table as it sets the bracket down to push a book) or other state variables due to the robot itself are ignored. We acknowledge that these thresholds are still quite high and lead to false negatives during contact detection. Since the spectators wouldn't touch the robot, however, we opted for more false negatives than false positives.



Shortcomings, Outstanding Items, and Lessons Learned

Some of our challenges were a result of weight. The shoulder motor has a fixed torque capacity, which limits the maximum weight it can support. This required us to minimize the distance between the load (books) at the tip and the shoulder motor. This resulted in shorter links and a smaller workspace. Furthermore, the system would oscillate after turning because of the weight, forcing us to slow down the movements. Reinforcing the links, either through a different design or material, may have improved this issue.

There's also a faulty USB 3 cable in the lab that caused us to have trouble connecting the RealSense Camera to the NUC.

There’s plenty of room for improvement, such as being able to reorient books at angles greater than +/- 180 degrees with respect to the x-axis, detect books without Aruco markers, and determine whether books are stacked upon each other or not. However, we are pretty satisfied with how the robot turned out and are glad we pursued this challenge.