1. Overview

The fundamental goal of this project was to create a robotic manipulator capable of cutting produce in an at-home setting to aid in personal food preparation. Initial designs called for a five degree of freedom (DOF) robot equipped with a small knife as its end effector capable of following a reciprocating trajectory to accomplish the cutting action. This manipulator was to be controlled using independent motion and force control, with performance measured by its ability to slice a hard boiled egg. To reduce overall complexity, this design was changed to a planar 4 DOF manipulator controlled primarily through motion control, with potential force control implemented solely for the cutting trajectory. The performance metric in this instance was also simplified from slicing an egg into multiple segments to only being halved.

The final, resulting goal of the project became using the simplified planar 4 DOF robot controlled via motion control only. Variations on gain tuning were also accomplished in order to compare the capabilities of the controller and its efficacy for the chosen task. In this case, the desired goal was to slice both a hard boiled egg and a banana into multiple segments and move the pile to the edge of the cutting surface. The following sections detail the specifics on how this was accomplished.

2. Literature Review

Automation has permeated the food industry significantly in the last decade. Whether it be improved smart irrigation systems, robotic fruit pickers, or automated vending machines, robots play an ever-increasing role in food preparation. Recently, more robots have been developed specifically for use alongside or in place of chefs. For instance, the Robotic and Mechanisms Lab (RoMeLa) at UCLA unveiled the “Yummy Operations Robot Initiative” or YORI, which serves as a chef in an automated kitchen, capable of creating a variety of dishes (Ackerman, 2023). This design attempts to bridge the gap between a fully automated system- in which food can easily turn out bland- and a robot working in a regular kitchen- which is difficult to design. Another area of research in the automation of food preparation comes in the form of RoboChop, a proposed autonomous slicing system capable of cutting a variety of produce on a single cutting board (Dikshit, 2024). While a physical prototype has not yet been developed, researchers have created motion planning algorithms using a wrist-mounted vision system intended to avoid obstacles, optimize cutting lines, and distinguish between whole and sliced produce. In a similar approach as RoMeLa’s YORI, Miso Robotics’ has created Flippy, a 6 DOF manipulator intended to work in a commercial grill setting preparing food like burgers and fries (Miso, 2024). In contrast to RoMeLa, Flippy uses specialty interchangeable end effectors and does not rely on an automated kitchen to work in conjunction with, operating in a normal kitchen used by humans. While the previous examples all involve the use of automation in a commercial kitchen setting, recent developments have also seen robots used in personal kitchens. A prime example of this is Moley’s Robotic Kitchen, which has various models that all center around a robot with one or two manipulators capable of performing a wide variety of cooking-related tasks (Moely, 2019). These systems can even be integrated with smart devices such as refrigerators to also notify users of when their pantry needs to be restocked.

Each of the examples above provide evidence to the power and capability that automation has in kitchens. However, it should be noted that with the exception of the Robotic Kitchen, these designs are primarily targeted towards businesses for reducing worker costs in a commercial setting. And while Moley’s system is a fully integrated at-home automated kitchen, upfront and installation cost make this format inaccessible for the majority of home chefs. The design presented in this report aims to target those individuals who are looking for a compact, relatively inexpensive aid in food preparation. It should be noted that the results of this process serve as a prototype for a future model capable of a wider range of tasks with improved sensing technology.

3. Controller Design Objectives

Throughout the project, three different types of controller architectures were implemented on the manipulator, all falling under the category of decentralized control. First, PID control with position feedback was implemented. This basic controller allowed for troubleshooting connectivity issues with the dynamixels, as well as pose maintenance which became integral in defining a homing script to minimize initial disturbance when following a trajectory. The position controller also allowed for pseudo-trajectory tracking, in which distinct waypoints were defined that the manipulator arbitrarily moved between. This was useful in ensuring that the arm could reach the waypoints that were later used in the complete trajectory tracking function

The next controller that was implemented on the robot was a decentralized PID with position and velocity control, which was used in the final product of the device. As part of this architecture, a detailed trajectory generation function was created such that both a desired position and velocity were accounted for in error terms. This resulted in a smoother motion as the manipulator transitioned between the via points that had been predetermined by the previous controller. This design also allowed for the knife to achieve a slicing motion rather than a purely rotational one, improving the success rate of cutting produce.

One final controller that was briefly implemented on the manipulator arm was decentralized control of position and velocity with a feedforward action of acceleration. The addition of the feedforward action was relatively straightforward as the trajectory generation function developed in the previous controller also calculated a trajectory for acceleration. However, this term had to be scaled by some gain for each controlled direction in operational space in order to try to improve performance. It was for this reason that this particular type of control was removed from the final design, as tuning the PID gains and individual feedforward terms became too complex. Ultimately, the feedforward action did help in further smoothing the robot’s motion along its trajectory but did so in a manner that slowed the movement such that it was never able to reach each waypoint, resulting in jerks and instability in between trajectory segments.

4. Methods

4.1 Robot Design

Figure 1 shows the Culinary Helper of Produce Preparation, or CHOPP, and its frame assignments. The robot consists of three links, actuated by four motors. Motors 1 and 2 control the position in the x and y directions, motor 3 controls the orientation ϕ of the knife about the Z3 axis, and motor 4 controls the knife angle Ψ about the Z4 axis. Note that due to geometric offsets the end-effector's x and y locations are also affected by ϕ, and Ψ also controls the end-effector position somewhat. Figure 2 also shows the physical assembled model of CHOPP in its intended experimental setup during testing.

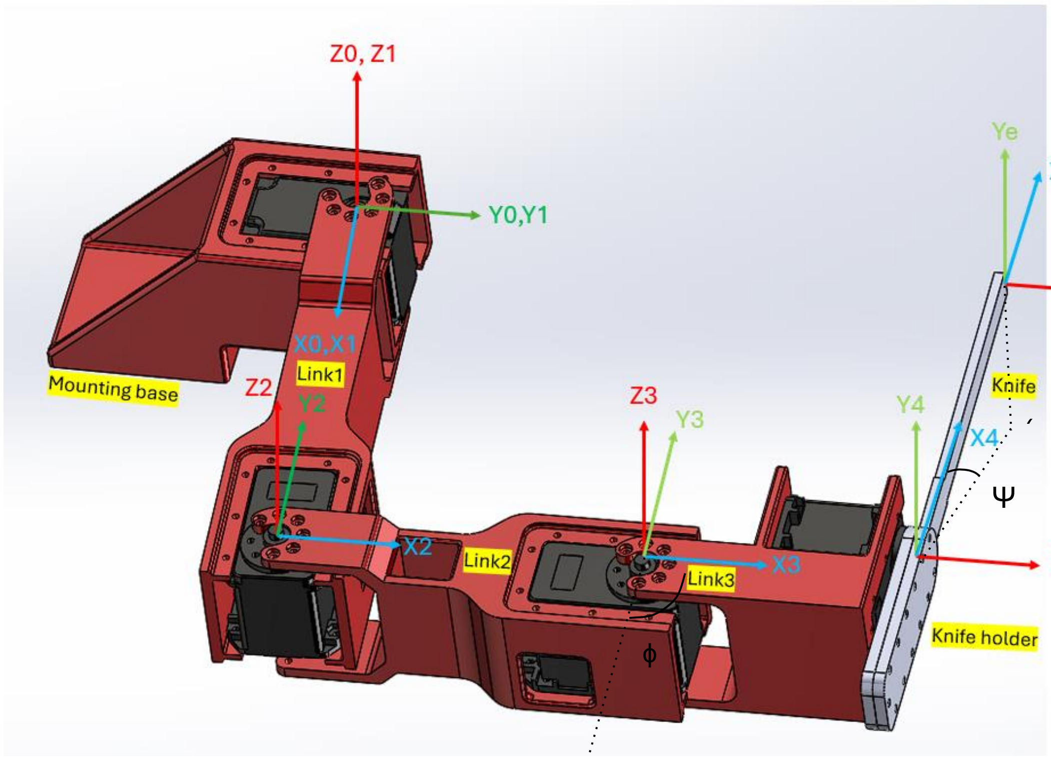


Fig. 1.1 Robot visualization and frame assignment

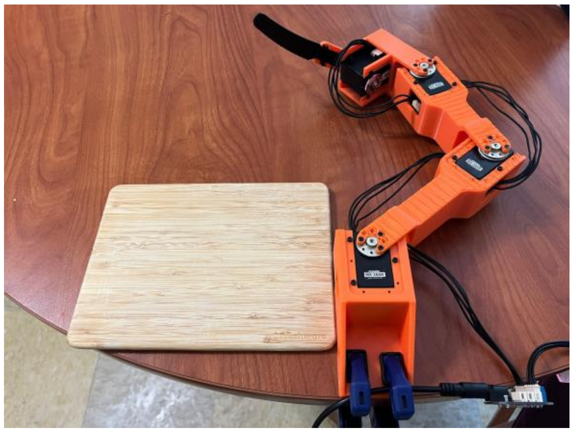


Fig. 1.2 Assembled Robot visualization

4.3 Trajectory Generation

The main purpose of CHOPP was to cut produce into slices. After the overall approach was finalized, trajectory positioning was limited to the XY plane and slicing action was limited to the knife rotation about Z3. Since the controller was set up to work in operational space, it was easier to intuit where to place the end effector in the global frame. Measurements of the cutting board and approximations on the average size of an egg informed the location of necessary via points, as well as the reachable workspace of the manipulator. To create the trajectory, basic position control was used to define waypoints for a position at home, above the produce, at the start and end of each slice, between slices, before sweeping, and after sweeping occurred. Between these waypoints, a cubic function was solved using a built-in Python function such that initial and final positions were met with no initial velocity for each controlled DOF. For the slicing action, a parabolic function was identified and chosen as it somewhat resembles the trajectory of slicing as done by a person. Since the actual slicing action happened in the YZ plane, a linear motion was defined against time in the Y direction, while a parabolic function for Z was created dependently. The initial and final z positions were defined and the slopes were equated to 0 at the start and end of each slicing action. As part of the controller metrics, the overall time of the action was limited to 10 seconds per slice.

Fig. 4 shows the planned trajectories of each operational space variable. During the first three seconds, the robot moves from the home position towards the produce. The knife is then positioned over the produce, after which it moves in the parabolic path to achieve the cutting action. One of the three instances of slicing occurs from 4 to 7 seconds, as seen in the parabolic descent of Ψ. After each cut is complete, the knife is held horizontally as the produce is moved along the x direction, as seen from 21 to 24 seconds, before returning to the home position.

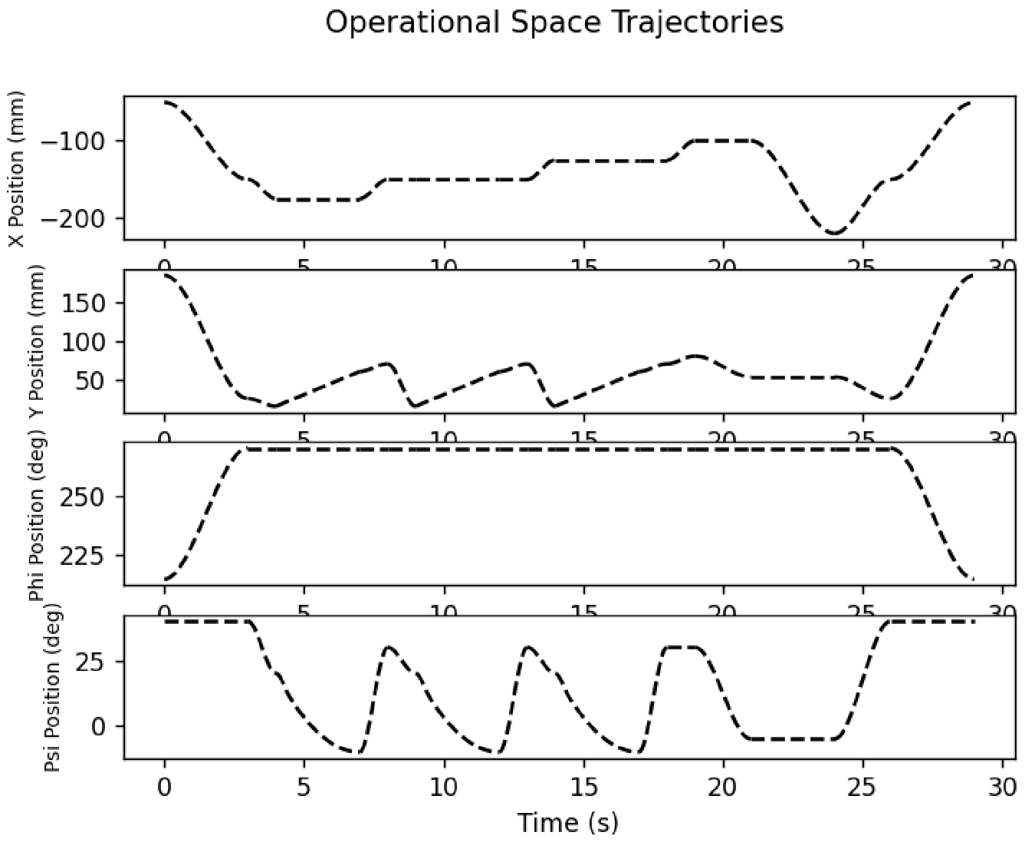


Fig 4: Planned Operational Space Trajectories

4.4 Controller Design Requirements

The following diagram outlines the block diagram of the controller in joint space. Note that there are no disturbances involved and that the transducer gains are set equal to one.

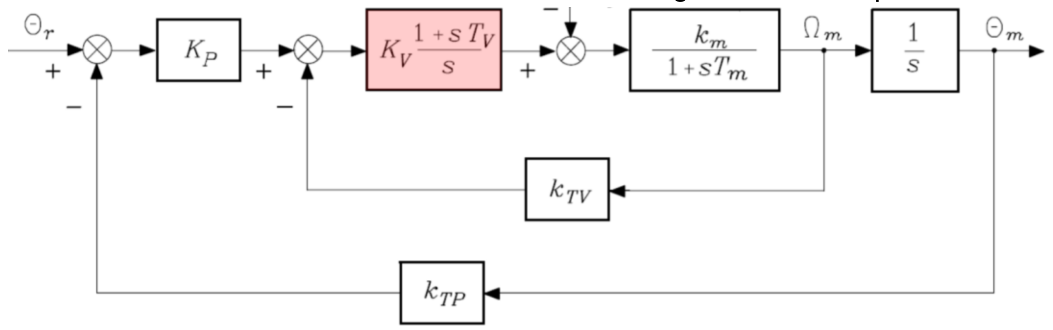
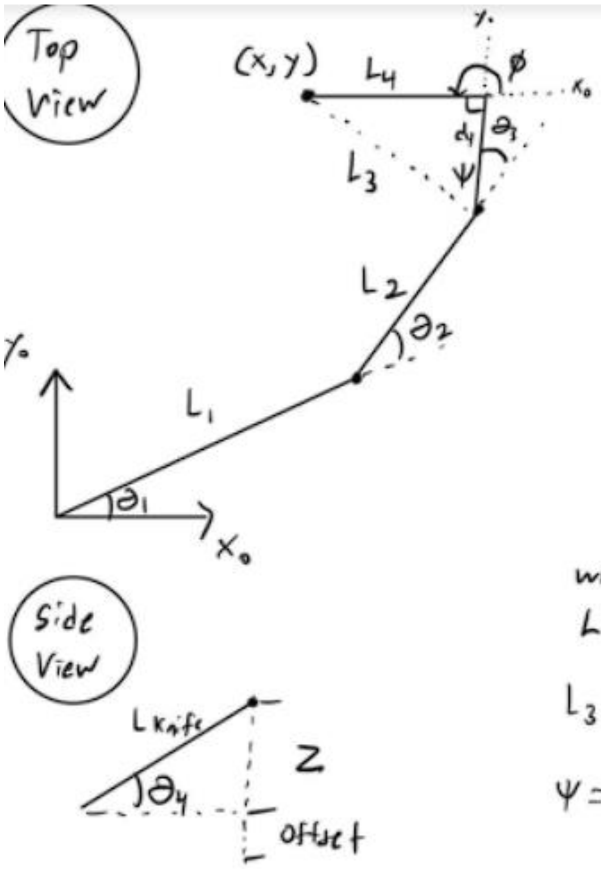


Fig. 5 Controller Block Diagram

In order to transform this control scheme into operational space, it was first necessary to calculate the forward kinematics in order to turn joint space outputs from the motors into operation space locations to be used for error determination. Using a geometric approach upon observing the system diagram shown in Fig. 1, the following equations were derived.



Forward Kinematics

$$x = L_1 c_1 + L_2 c_{12} + L_3 c_{123\psi}$$

$$y = L_1 s_1 + L_2 s_{12} + L_3 s_{123\psi}$$

$$z = L_{knife} s_4 + offset$$

$$\phi = \theta_1 + \theta_2 + \theta_3 + \frac{\pi}{2}$$

where:

$$L_4 = L_{knife} c_4$$

$$L_3 = \sqrt{d_4^2 + L_{knife}^2}$$

$$\psi = \text{atan2}(L_{knife}, d_4)$$

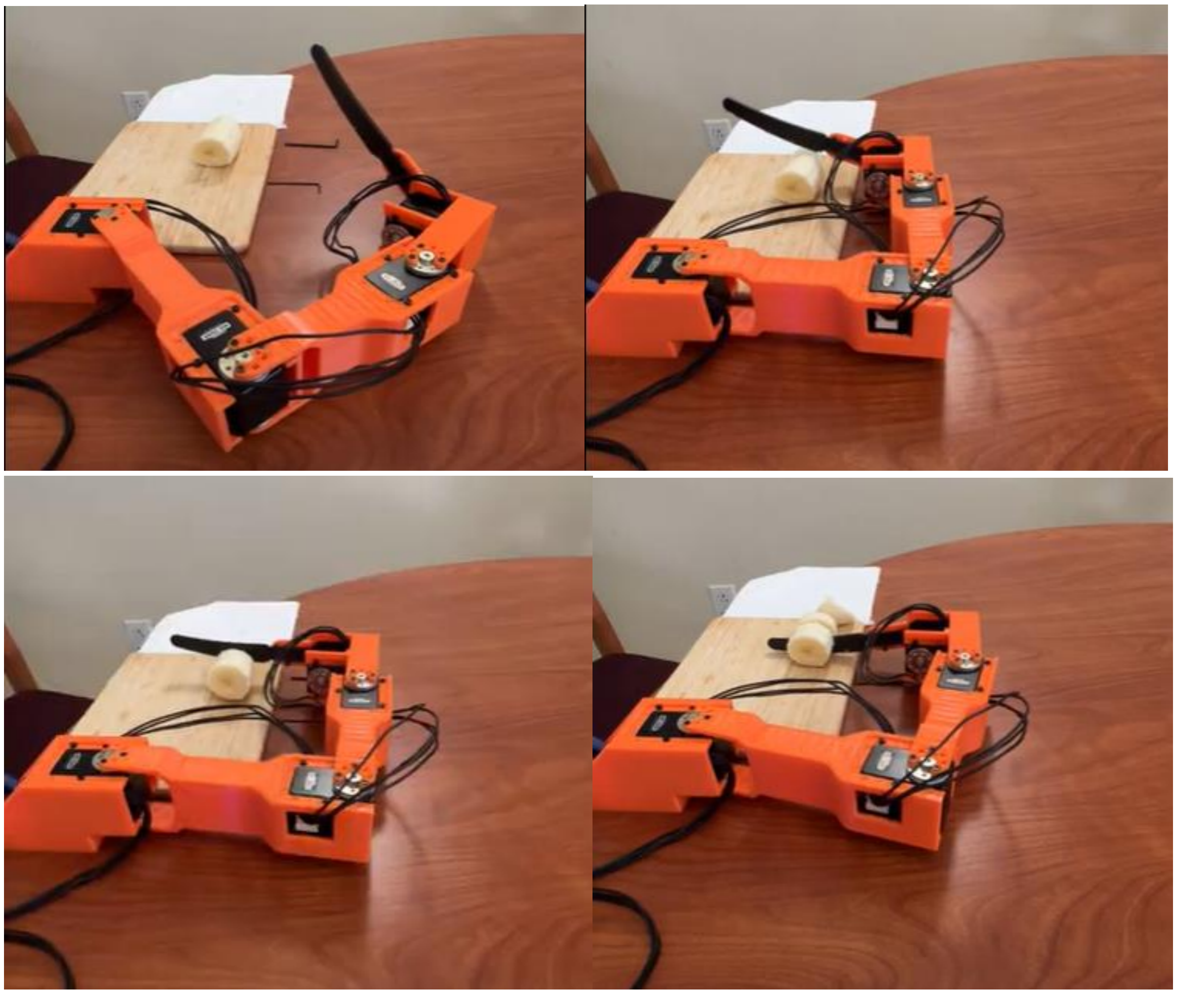
The analytical Jacobian and its transpose were used to convert joint velocities to operational space and to convert operation space errors into joint errors to be used as a control input respectively. Since the position and orientation of the knife were all controlled by rotations about parallel axes and the rotation of the knife was independently controlled, the derivation of the analytical Jacobian was simplified. The following matrix was used in conjunction with the previous equations to be evaluated at each time step.

In order to test how effective the controller was at performing the desired test, four metrics were evaluated during each test. First, the overall time taken to complete the entire trajectory should be 10 seconds per each desired cut. This includes extra motion to and from the home position, as well as the motion to sweep the food off of the cutting board. The second metric evaluated was that the food should be able to be placed within the knife's reach in an arbitrary orientation and still be successfully cut. The next metric is ensuring that the cutting surface and manipulator does not sustain any damage during operation, and that the cutting board does not get bumped or shifted during tests. The final metric used to evaluate the performance of CHOPP was that the produce should only come in contact with the plastic knife, and not any other part of the manipulator, to ensure food safety.

In order to create these performance metrics, several assumptions were made based on the chosen controller architecture. One implied assumption that was held due to the nature of decentralized control was ignoring gravity, friction, and inertial terms. The first asserted assumption is that any round produce will be cut in half so there is a flat surface in contact with the cutting board, increasing friction and preventing slips. As seen in the following sections, this was not strictly necessary as the knife provided enough downward force to keep the food firmly in place on the board. The other, more important assumption that was made to justify the use of decentralized PID control was that the knife was sharp enough and produce soft enough that external forces could be ignored. This is because decentralized control is intended for operation in free space and is not particularly adaptable to disturbances. The validity of this assumption is also made in the ensuing sections.

5. Results

The following series of images demonstrates CHOPP as it works to cut up a portion of a banana before pushing the slices off the edge of the cutting board.



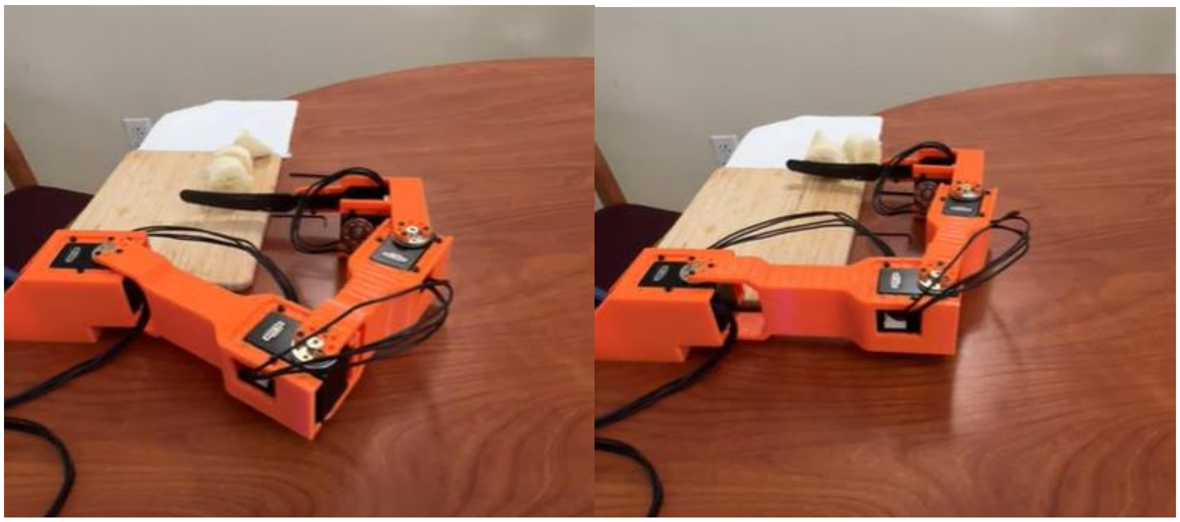


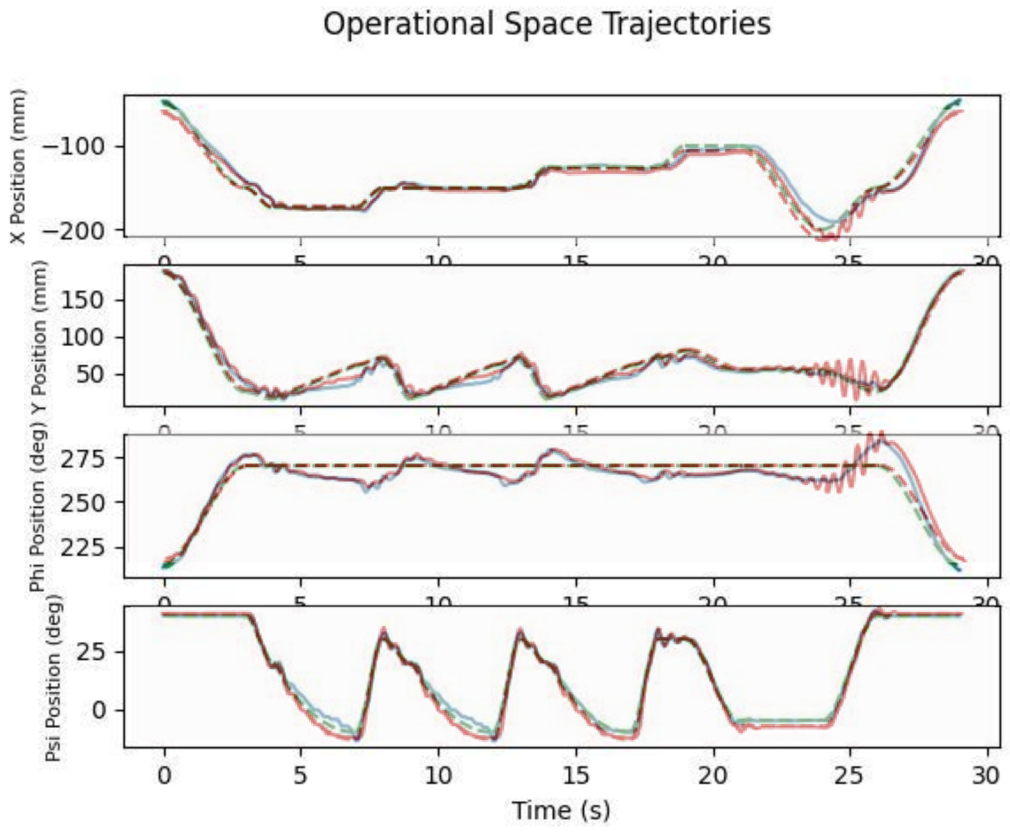
Fig 6. Time snapshots of C.H.O.P.P cutting a portion and sliding

Performance Assessment

Overall, CHOPP performed its intended functions successfully, though the way it went about doing so was not necessarily accomplished as intended. Fig. FILL shows the performance results of each controlled DOF in operational space when no produce was present. Note that the actual positions are shown in solid green while the desired trajectory is shown dashed in blue as in Fig. FILL. As can be seen from the graphs, the x, y and Ψ trajectories tracked fairly well, while the ϕ trajectory had a somewhat difficult time keeping the knife position downard in the y direction (at 270 degrees). This was to be somewhat expected due to geometric offsets and interconnectivity between the y and ϕ directions especially.

Figs. FILL and FILL show the performance of the manipulator when actually slicing through a banana and egg respectively. Firstly, it can be seen that between each of these tests- as well as the free motion trajectory discussed above- the performance in the x-direction does not vary significantly. In the y-direction however, it can be noted that while performance remains relatively strong when operating in free space, the end effector is not able to completely follow the slicing motion when it is cutting the food from 4 to 20 seconds. This is because the food, and to some extent the cutting board, in addition to the correlation between the z and y trajectory during the slice, prevent a smooth linear motion along the y axis. It should be noted however, that the end effector does reach the requisite y location at the end of each slice when the error becomes high enough. Furthermore, this discrepancy is exacerbated when slicing the egg due to increased friction from the yolk and a lowered final slicing angle. Examining the orientation ϕ of the knife, the overall performance when physically cutting food actually improved, the orientation is able to remain more stable. However, there is more oscillation in the banana and egg which is a result of the knife coming into contact with the substrate and attempting to swing further from its desired orientation before being stopped by the resistance of the food. Finally, while the knife angle Ψ still tracks very well in both cases, there is more oscillation when the knife is raised and returns to free space, and the effects of the necessary integral action can be seen as the knife approaches its final cutting angle in a more linear trajectory than desired. It should also be noted that the gains between these trials had to be updated somewhat so that cutting could be accomplished, as will be compared in the next section.

One final metric that is useful in evaluating the performance of the controller is its operation in free space compared to interacting with an object. Fig. FILL shows both trials of operation in free space and when cutting a banana, using the same gains as described in the following sections. Note that due to a lack of data logging, this figure was created using an overlay of two separate graphs. Based on the comparison, it is clear that the increased gains used for the banana cutting caused more oscillations when used in free space, most notably in the y and ϕ directions around 25 seconds. This highlights the fact that due to limitations in the decentralized control, a balance needs to be made between having low enough gains to maintain stability and high enough gains so the knife can fully cut through the food.



6. Discussion

6.1 Gain Insights

Table one shows a comparison of the diagonal entries of the gain matrices used to successfully slice a banana and egg, as outlined in Fig. FILL. Note that the vast majority of the gains used were identical, with two exceptions. Firstly, the K_P value controlling the knife angle was increased somewhat, which was necessary to allow for enough force to be generated to fully cut through the egg, which had a tougher consistency than the banana. Additionally, the K_I value for the knife angle was increased by 25% for the egg, also to allow for the egg to be completely cut into pieces and not merely cut partially. It should also be noted that, as mentioned previously, the desired final angle of the knife was set to be five degrees lower in the egg trial compared to the banana. This artificial increase in error was done in order to avoid increasing the gains on the knife angle further, which resulted in instability. Another alternative would have been to increase the cutting time to allow for more integral action to take effect, but this was avoided in order to maintain the metric of 10 seconds per slice. In general, the thing of the x direction remained relatively easy, most likely because the knife was oriented along the y axis. This mean that a small change in the y or ϕ direction directly contradicted each other, hence the imperfect trajectory tracking discussed previously.

Gain	Banana	Egg
K_P	[260e-7, 180e-7, 180e-5, 235e-4]	[260e-7, 180e-7, 180e-5, 240e-4]
K_I	[90e-8, 100e-8, 180e-5, 400e-4]	[90e-8, 100e-8, 180e-5, 500e-4]
K_D	[40e-10, 190e-10, 170e-7, 35e-5]	[40e-10, 190e-10, 170e-7, 35e-5]

6.2 Controller Comparison

The following describes the insights into each members' way of tuning and how well their gain choices satisfied the controller metrics defined previously.

Divik

X and Ψ are tracked relatively well, with the x-location primarily affected by the first three gains in each direction. Over-tuning the x-location affects the ability of the manipulator to achieve accurate y and ϕ positioning. Tuning the second and third gains of y and ϕ was made difficult by the fact that both are strongly coupled finding a balance between the two without significant oscillation was challenging. Any oscillations on any motor led to a cascading effect both upstream and downstream as the manipulator worked to compensate for disturbances. Overall, increasing the K_I gains helped reduce the phase difference in tracking, particularly during the straight line motion. However, increasing K_I too much resulted in oscillations and overshoot.

David

By working to increase the gains, along with the controller's operating frequency, the total time for moving to and from the home position, slicing three times, and piling pieces off the cutting board was accomplished in 30 seconds. This satisfies the metric that for each slice, the entire operation should take 10 seconds. While variations in the size and position of the egg yolks made the orientation of each egg important to slicing, variation in the curvature and size of each banana showed the robot's ability to cut in any orientation provided uniform consistency. While cutting through the egg was possible, it became a balance of increasing gains to allow for full cutting without sacrificing too much stability during motion in free space. Even though the entire arm experienced some torsion whenever the knife came into contact with the produce or cutting board, it fell within the material compliance of the structure and did not damage the system. Furthermore, the cutting surface remained undamaged by the knife and the board remained undisturbed after enough gain tuning was accomplished. In regards to the final controller metric, the knife holder did come close to the produce upon a successful cut but only the knife made contact with the food. That being said, this was highly dependent on where the object was placed, as distance further from the base of the knife did not allow for enough torque to be generated by the last motor while distances too close to the based of the knife resulted in the food getting squished under the knife holder and occasionally dragged off the edge of the cutting board. This could have been improved by better markings made on the cutting surface or by implementing a vision system, as will be discussed later. In terms of general tuning, the x direction and knife angle were the most independent DOFs so they were tuned first. Then a balance of tuning occurred to provide reasonable performance between the y and knife orientation directions. On a final note, the controller did not perform well when large changes in knife orientation were required alongside changes in x and y location. This is most evident from 24 to 26 seconds- when the manipulator reaches the cutting via point before returning home- where the largest error in knife orientation can be observed.

Atharva
Tuning the gains was the most time consuming part of the project for attaining the smoothness in the robot's movement. The tuning was a 2 step iterative process with first tuning the orientation motors and then the position motors. Decoupling the motors was easier for the 1st and last motor. The last orientation motor was the easiest to tune as seen in the figure below. The actual and expected results were closely matched. There were some minor oscillations at the peaks but they were for a very short duration and did not affect the operation in any way. Tuning the robot for X was relatively simpler than for Y position and Phi orientation. Motors 2 and 3 were highly coupled as a small change in gains on either of the 2 had a direct impact on the other, hence finding a sweet stop was relatively difficult. One key observation was motors with high degree of movement had larger gain windows compared to the motors which moved relatively less as a small change caused the motor to flip. Coming to the actual gain values, the gains had to be very low in the order of magnitude of 10^-7. Comprehensive gain values are listed in the table below. Kp and Kd values were easier to visualize as small increments were evident from the trajectory data. On the other hand, the effect of Ki was only seen after increasing it in magnitude. Large Ki gains made the actual trajectory follow closely to the desired one and had the largest impact where the trajectory was a straight line.

Overall, the gains varied between each individual for a variety of reasons. As Table 2 shows, the gains had to be reduced for Divik and Atharva relative to David. This is primarily due to latency issues as the latter was able to allow for faster serial communication and control frequency. The effects of this can also be observed by a comparison of Figs. FILL to FILL, which shows that David's testing resulted in a much smoother trajectory with minimal overshoot and oscillation. Another factor contributing to the oscillations of Divik and Atharva's tests was the fact that no produce was cut during their trials. Thus, tuning gains too high led to increased instability in free space relative to interacting with an external force, as described previously.

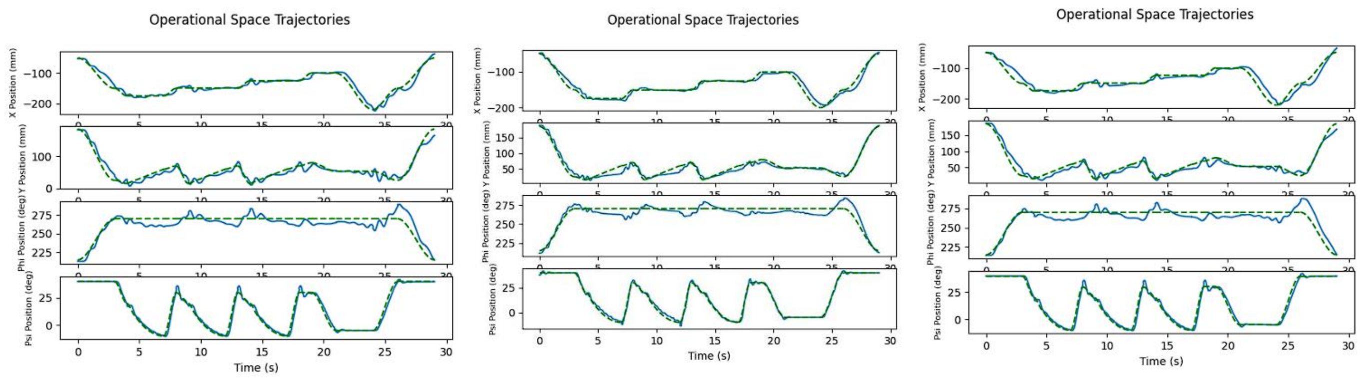


Fig. 7. Comparison of position tracking (Divik, David and Atharva in order)

6.3 Gain comparisons

Gain	Divik	David	Atharva
K_P	150e-7, 180e-7, 180e-5, 23.5 e-4	[260e-7, 160e-7, 150e-5, 200e-4]	190e-7, 110e-7, 130e-5, 120e-4
K_I	400e-7, 500e-7, 600e-5, 600e-4	[90e-8, 100e-8, 180e-5, 100e-4]	900e-7, 900e-7, 900e-5, 600e-4
K_D	110e-10, 115e-10, 170e-7, 35e-5	[40e-10, 190e-10, 180e-7, 35e-5]	120e-10, 125e-10, 600e-7, 108e-5

6.4 Capabilities and Limitations

In theory, CHOPP is capable of performing various styles of vertical cuts for a wide range of foods. However, the choice to implement a decentralized controller set limitations on the feasibility of actually succeeding in these operations. Particularly in free space, the decentralized control allowed for relatively easy trajectory adaptation, provided that via points were well defined and determined previously. By using the trajectory function found in the Python code (see Appendix), the number of slices and style of cutting was easily adjustable, as were the coordinates of any other particular point. Also adjustable using the trajectory function was the speed each segment of the trajectory was completed in. This allowed for operating cycles to be reduced from 45 seconds to 30 seconds, fulfilling one of the previously defined controller metrics. This was all contingent on proper tuning, which began with very conservative values to avoid instability and damage but were later increased to improved consistency, speed, and overall performance. Increasing the gains also allowed for more refined slicing trajectories to be generated with higher resolutions, improving the smoothness of the cutting action.

While the implemented decentralized control was successful at performing the above operations, it was limited in several capacities due to the nature of the control structure. While the overall cycle time was able to be reduced by 33%, this was mainly due to reductions in time during the free motion portion of the trajectory. Since each motor was essentially operating independently of each other- with external forces, inertial terms, and coriolis effects ignored as described earlier- attempting to speed up the motion further resulted in wild instability. The deficiency of this controller was most evident during the actual slicing of the produce. Recall that one assumption used when validating the choice of controller was that the knife was sharp enough and the egg and banana were soft enough that external forces could be ignored. While this assumption somewhat held up, it was not fully realized since gains had to be increased for the cutting action, as mentioned earlier. This meant that from the controller's perspective, the entire trajectory could not be completed in free space. This is further evidenced by the fact that error values had to be artificially raised by commanding the end of the slicing trajectory to be at a point beneath the cutting board. As seen from the comparison of banana to egg performance, variability in the density of the substrate being sliced mandated the stability of the end effector when operating in free space. This shows that the assumption of ignoring external forces can be reasonably accepted only for particular, uniform types of food.

6.5 Challenges

There were several challenges that were necessary to overcome during the duration of the project. As mentioned previously, finding an appropriate balance between tuning gains in the y-direction and z-rotation was difficult. Furthermore, the magnitude that each gain should be tuned around also took time to determine, as the output of encoder readings and input range of PWM signals were not immediately intuitive. Another challenge that occurred during testing was occasional power interruption to the motors, causing them to continue to respond to whatever PWM signal last received until a power cycle could be completed. A faulty main power cable was the suspected cause, but since each motor was chained together the fault could have also lied in the connecting lines. The majority of these cables were replaced or re-plugged, though the design of the linkages made disassembly somewhat difficult. Another issue that affected the manipulator's ability to accurately follow trajectories of increasing speed was a latency setting embedded in the serial communication that was used to send and receive information from the motors. Luckily, teaching assistant Cole Ten aided in the solution to this problem by explaining how to reduce the embedded latency from 16ms to 1ms, allowing the controller to operate at 50 Hz rather than 30. One of the largest challenges involved with the project was the fact that egg yolks have a much thicker consistency than egg whites when boiled and would often stick to the knife. Furthermore, each egg cooked differently, resulting in variations in yolk location and size. This somewhat limited the orientation that the egg could be positioned in, since a cross-section of an egg with too high of a yolk percentage would not be fully cut. When the yolk was removed, the remaining hollow structure of the egg could not keep its own shape, and the knife action only succeeded in deforming the remaining egg whites rather than slicing them. As a partial solution, bananas were introduced as a secondary product to cut, with the benefits of a uniform consistency and low friction. The variation in size and curvature between bananas also showcased the adaptability of the manipulator to cut varying orientations of food. As an added benefit, the cost required to test the device decreased as bananas are significantly cheaper than eggs.

6.6 Future Direction

There are several ways that this prototype could be expanded upon in the future. Starting with hardware, the base design could be remodeled to flare outward in order to provide better stability in more directions. While it did not impact performance for this project, the current base design does the most to prevent the arm from bending for points along the global y axis. Another hardware redesign would be changing the geometry of the 3D-printed links in order to increase ease of the assembly and allow for a greater range of motion, as there were physical interactions that provided hard limits on joint angles. The end effector could also be redesigned in a manner to reduce the offset of the knife relative to the wrist, which would aid in tuning. Finally, utilizing a sharper knife or swapping out the tool for something lightly oscillatory similar to a jigsaw would increase the validity of the assumption that external forces can be ignored.

Other improvements can also be made on the software side of the project. Most importantly, the raw data for each trajectory could be saved from each trial to a csv file, for instance. This would allow for a better comparison of the data and provide better record-keeping for future users. Another feature that would greatly increase the applicability of the project would be incorporating a vision system. This would allow the user to arbitrarily place their produce on the cutting surface before letting CHOPP run its task. In addition to the software required to convert camera images to waypoints, the software would also need to include an updated trajectory generation function for online path creation. Finally, several features could be added to improve the user interface, such as how many slices they would like or the thickness of each cut.

One final area of the project that could be improved is the controller in general. As mentioned previously, attempting to implement feedforward action to the decentralized PID controller led to instability. The gains applied to the feedforward terms for each direction could be better tuned in an attempt to improve performance. Another improvement to the current type of controller would be to incorporate some kind of gain scheduler, which would be able to increase the controller gains before the actual slicing trajectory occurs. This would allow for smooth motion in free space when transitioning between waypoints while allowing for sufficient torque to be generated so that produce could be cut fully. One final way to improve the controller design would be to incorporate force control. Unless improved motors were purchased, using an indirect force controller like impedance control, which would be used to apply a desired force through the produce until reaching a threshold resistive force against the cutting board once the cut is complete. Overall there are several different ways that the hardware, software, and control design could be improved to increase performance of the chosen task.

7. Bibliography

Ackerman, Evan. 2023. “YORI: A Hybrid Approach to Robotic Cooking - IEEE Spectrum.” Spectrum.ieee.org. September 26, 2023. https://spectrum.ieee.org/romela-cooking-robot.

Bartsch, Alison, Atharva Dikshit, Amir Fariman, and Abraham George. n.d. “RoboChop: Autonomous Framework for Fruit and Vegetable Chopping Leveraging Foundational Models.” Ar5iv. Accessed May 13, 2024. https://ar5iv.labs.arxiv.org/html/2307.13159v1.

Mei, Song, Fengque Pei, Zhiyu Song, and Yifei Tong. 2022. “Design and Testing of Accurate Dicing Control System for Fruits and Vegetables.” *Actuators* 11 (9): 252. https://doi.org/10.3390/act11090252.

Miso Robotics. n.d. “Flippy.” Miso Robotics. Accessed May 13, 2024. https://misorobotics.com/flippy/.

Moley Robotics. 2019. “Moley – the World’s First Robotic Kitchen.” Moley.com. Moley Robotics. 2019. https://www.moley.com/.

8. Appendix

8.1 Contributions

Divik: CAD modeling, slicing trajectory generation, forward/inverse kinematics, individual tuning, final presentation, and report writing.

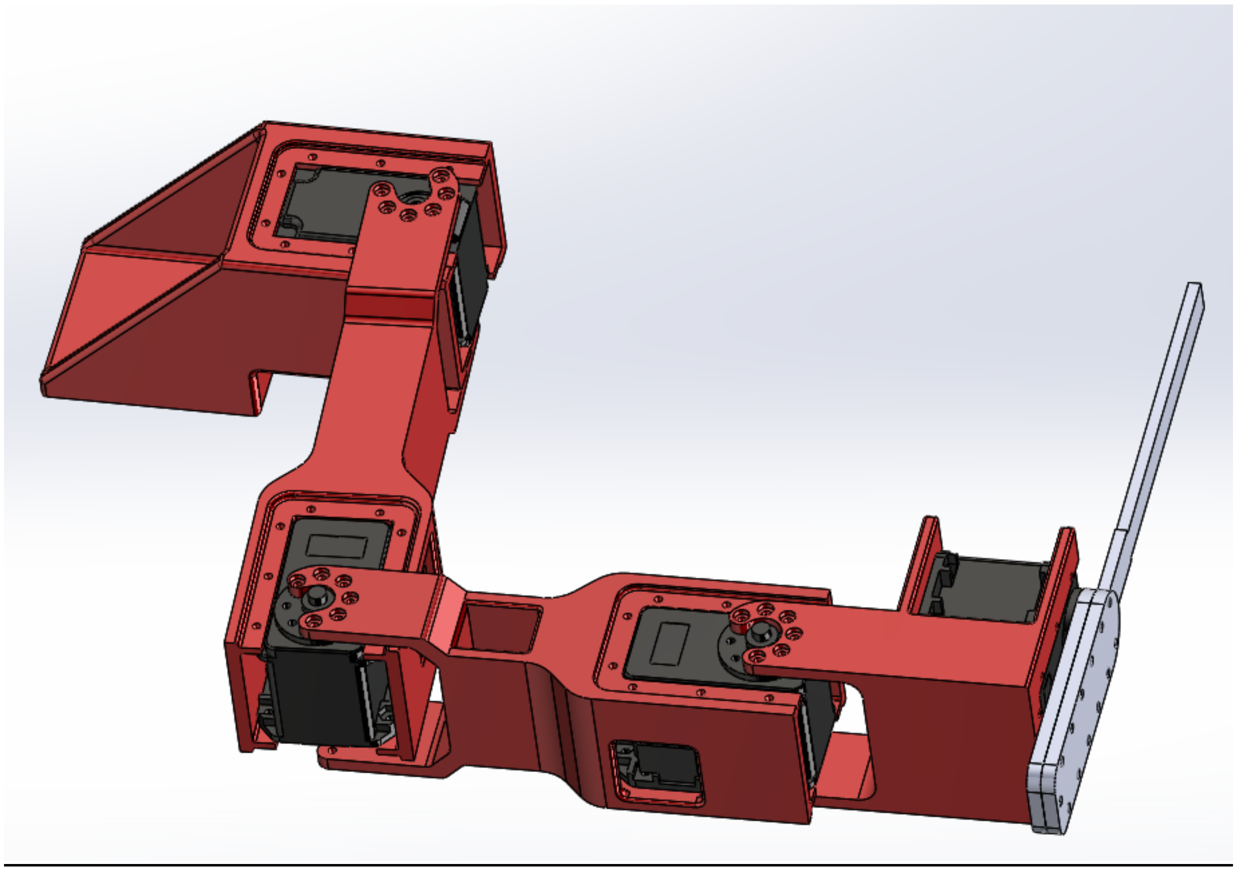
David: Proposal writing, dynamixel interfacing, slicing/3D printing, wiring/assembly, controller writing, progress-presentation creation, non-slicing trajectory generation, general tuning, individual tuning, egg/banana testing, final presentation, and report writing.

Atharva: CAD modeling/improvements, forward kinematics, trajectory generation, individual tuning, final presentation, and report writing.

8.2 Bill of Materials

Component	Unit Cost	Quantity	Total Cost
U2D2	$25	1	$25
Dynamixel MX-28AR	$300	4	$1,200
Power Hub	$12	1	$12
Clamps	$6	2	$12
3D Printed PLA	$0.10/gram	248g	$24.8
Cutting Board	$10	1	$10
Plastic Knife	$0.05	1	$0.05
Total			$1,283.85

8.3 Solidworks CAD Designs



8.4 Python Code

The following is a complete copy of the Python code used to accomplish the cutting and other motion seen in the final product.

```python
# Import necessary Python libraries
# Opertational
import math
import signal
import time
from collections import deque
from collections.abc import Sequence
import numpy as np
from numpy.typing import NDArray
import matplotlib.pyplot as plt
from scipy.interpolate import CubicSpline
# Dynamixel
from dynamixel_utils.dc_motor_model import DCMotorModel
from dynamixel_utils.motor import(
    get_mx28_control_table_json_path,
    group_sync_write,
    group_sync_read,
)
from dynio.dynamixel_controller import DynamixelIO, DynamixelMotor
from dynamixel_utils import FixedFrequencyLoopManager


# Defining function to get trajectory
def getTraj(t_start, t_end, n, x_start, x_end):
    # Defining actual and trajectory timesteps
    times = np.linspace(t_start, t_end, n)
    intervals = np.linspace(t_start, t_end, n)
    # Calculating directional trajectories
    points = [[], [], [], []]
    for i in range(4):
        a_0 = x_start[i]
        a_2 = 3*(x_end[i] - x_start[i])/(t_end**2)
        a_3 = 2*(x_start[i] - x_end[i])/(t_end**3)
        points[i].append(a_3*np.power(intervals,3) +
a_2*np.power(intervals,2) + a_0)
    # Reformmating directional trajectories
    points = np.stack((points[0][0], points[1][0], points[2][0],
points[3][0]), axis=0)
    # Computing trajectory derivatives
    coordinates = CubicSpline(x=intervals, y=points, axis=1,
bc_type="clamped")
    # Returning position, velocity, and acceleration
    return intervals, coordinates(times), coordinates(times, 1),
coordinates(times, 2)


# Defining function to get cutting trajectory
def getTraj_cutting(t_start, t_end, tb, n, x_start, x_end):
    # Defining actual and trajectory timesteps
    times = np.linspace(t_start, t_end, n)
    intervals = np.linspace(t_start, t_end, n)
    interval_xyp = np.linspace(t_start, t_end, n)
    interval_z1 = np.linspace(t_start, tb, 2)
    interval_z2 = np.linspace(tb+0.01, t_end, n)
    interval_z = np.hstack((interval_z1, interval_z2))
    # Calculating directional trajectories
    x = np.linspace(x_start[0], x_end[0], n)
    y = np.linspace(x_start[1], x_end[1], n)
    psi = np.linspace(x_start[2], x_end[2], n)
    # Caclulating parabolic variation of z with time
    z = (x_start[3] - x_end[3])/t_end**2 * interval_z ** 2 - 2*(x_start[3]
- x_end[3])/t_end * interval_z + x_start[3]
    #z = np.rad2deg(np.arcsin(z/125))
    # Storing points
    points_xyp = np.stack((x, y, psi), axis=0)
    # Computing trajectory
    coordinates_xyp = CubicSpline(x=interval_xyp, y=points_xyp, axis=1,
bc_type="clamped")
    coordinates_z = CubicSpline(x=interval_z, y=z, axis=1,
bc_type="clamped")
    # Reformatting
    pos = np.vstack((coordinates_xyp(times), coordinates_z(times)))
    vel = np.vstack((coordinates_xyp(times,1), coordinates_z(times,1)))
    acc = np.vstack((coordinates_xyp(times, 2), coordinates_z(times, 2)))
    # Returning position, velocity, and acceleration
    return intervals, pos, vel, acc


# Defining Trajectory Class
class TrajectoryGeneration:
    # Defining initialization function
    def __init__(self, times: NDArray[np.double], coordinates:
NDArray[np.double]):
        super().__init__()
        # Initializing parameters to class
        self.times = np.asarray(times, dtype=np.double)
        self.coordinates = np.asarray(coordinates, dtype=np.double)
        self.ix: int = 0

    # Defining function to be executed when called
    def __call__(self, t: float) -> NDArray[np.double]:
        # Updates which timestep is accessed
        if t >=self.times[self.ix]:
            self.ix += 1
```

```python
        # Accesses the final timestep if the end is reached
        self.ix = min(self.ix, len(self.times) - 2)
        # Returns the coordinates at the current timestep
        return self.coordinates[:, self.ix]


# Defining Controller Class
class PDController:
    # Defining initialization function
    def __init__(
            self,
            serial_port_name: str,
            dxl_ids: tuple[int, int, int, int],
            K_P: NDArray[np.double],
            K_I: NDArray[np.double],
            K_D: NDArray[np.double],
            link_lengths = [137, 110],
    ):
        # Initializing parameters to controller
        self.K_P = np.asarray(K_P, dtype=np.double)
        self.K_I = np.asarray(K_I, dtype=np.double)
        self.K_D = np.asarray(K_D, dtype=np.double)
        self.link_lengths = link_lengths
        self.freq = 50.0
        self.period = 1 / self.freq
        self.integral_limit = 30

        # Ensuring proper communication
        self.loop_manager =
FixedFrequencyLoopManager(period_ns=round(self.period * 1e9))

        # Creating storage variables
        self.timestamps = deque()
        self.e_i = [0, 0, 0, 0]
        self.position_history = deque()
        self.location_history = deque()
        self.dx_history = deque()

        # Creating motors
        self.dxl_io = DynamixelIO(device_name=str(serial_port_name),
baud_rate=57_600)
        self.dxl_ids = dxl_ids
        self.motors = np.empty(len(self.dxl_ids), dtype=DynamixelMotor)
        for i in range(len(self.dxl_ids)):
            self.motors[i] = DynamixelMotor(
                dxl_ids[i],
                self.dxl_io,
                json_file=get_mx28_control_table_json_path(),
                protocol=2,
            )

        # Assuming each motor is identical
        pwm_limit_add, pwm_limit_len =
self.motors[0].CONTROL_TABLE["PWM_Limit"]
        pwm_limits = group_sync_read(self.dxl_io, self.dxl_ids,
pwm_limit_add, pwm_limit_len,)
        velocity_limit_add, velocity_limit_len =
self.motors[0].CONTROL_TABLE["PWM_Limit"]  #DOUBLE CHECK#####
        velocity_limits = group_sync_read(self.dxl_io, self.dxl_ids,
velocity_limit_add, velocity_limit_len,)
        self.pwm_limits = np.empty(len(self.dxl_ids))
        self.velocity_limits = np.empty(len(self.dxl_ids))
        for i in range(len(dxl_ids)):
            self.pwm_limits[i] = pwm_limits[dxl_ids[i]]
            self.velocity_limits[i] = velocity_limits[dxl_ids[i]]
        self.motor_model = DCMotorModel(self.period,
pwm_limits=self.pwm_limits)

        # Cleaning up
        signal.signal(signal.SIGINT, self.signal_handler)
        signal.signal(signal.SIGTERM, self.signal_handler)


    # Defining function to track pose
    def track(self, times, traj, dtraj, ddtraj, prev_start):
        # Resetting integral sum
        self.e_i = [0, 0, 0, 0]

        # Recording start time
        start_time = time.time()
        cur_time = start_time

        # Creating trajectory objects
        self.traj = TrajectoryGeneration(times, traj)
        self.dtraj = TrajectoryGeneration(times, dtraj)
        self.ddtraj = TrajectoryGeneration(times, ddtraj)

        # Continuing through trajectory
        while cur_time - start_time < times[-1]:
            # Step 0: Extract Waypoint
            cur_time = time.time()
            # Getting desired position and velocity
            x = self.traj(cur_time - start_time)
            dx = self.dtraj(cur_time - start_time)
            ddx = self.ddtraj(cur_time - start_time)

            # Step 1: Get Feedback
            # Getting joint position and velocity
            q_cur = self.getQ()
            dq_cur = self.getdQ()
            # Converting to operational space
            x_cur = self.FK(q_cur)
            dx_cur = self.J_A(q_cur) @ dq_cur.T

            # Step 2: Calculating Error
            e_p = np.subtract(x, x_cur)
            e_d = np.subtract(dx, dx_cur)
            self.e_i = np.add(self.e_i, e_p * self.period)
            # Capping integral error
            for i in range(len(x)):
                if self.e_i[i] < -self.integral_limit:
                    self.e_i[i] = -self.integral_limit
                if self.e_i[i] > self.integral_limit:
                    self.e_i[i] = self.integral_limit

            # Step 3: Calculating Control Action
            # Scaling by gains
            y = self.K_P * e_p + self.K_D * e_d + self.K_I * self.e_i
            '''# Feedforward Action
            scale = 0.0000005
            y += scale * ddx'''
            # Converting from operational space to control effort
            u = self.J_A(q_cur).T @ y.T
            # Converting to PWM
            pwm_command = self.motor_model.calc_pwm_command(u)

            # Step 4: Sending Command
            self.sendPWM(pwm_command)

            # Step 5: Updating and Storing
            self.timestamps.append(cur_time - start_time + prev_start)
            self.position_history.append(q_cur)
            self.location_history.append(x_cur)
            self.dx_history.append(dx_cur)
            self.loop_manager.sleep()


    # Defining function to turn motors on
    def torqueOn(self):
        for i in range(len(self.motors)):
            self.motors[i].torque_enable()
            print("Turned on Motor ", i+1)

    # Defining function to turn motors off
    def torqueOff(self):
        for i in range(len(self.motors)):
            self.motors[i].torque_disable()
            print("Turned off Motor ", i + 1)

    # Defining function to get current joint space pose
    def getQ(self) -> NDArray[np.double]:
        (cur_pos_control_add, cur_pos_len,) =
self.motors[0].CONTROL_TABLE["Present_Position"]
        joint_ticks = group_sync_read(self.dxl_io, self.dxl_ids,
cur_pos_control_add, cur_pos_len,)
        q = np.empty((len(self.motors),), dtype=np.double)
        for i, motor in enumerate(self.motors):
            q[i] = self.convert_encoder_tick_to_deg(motor,
joint_ticks[motor.dxl_id])
```

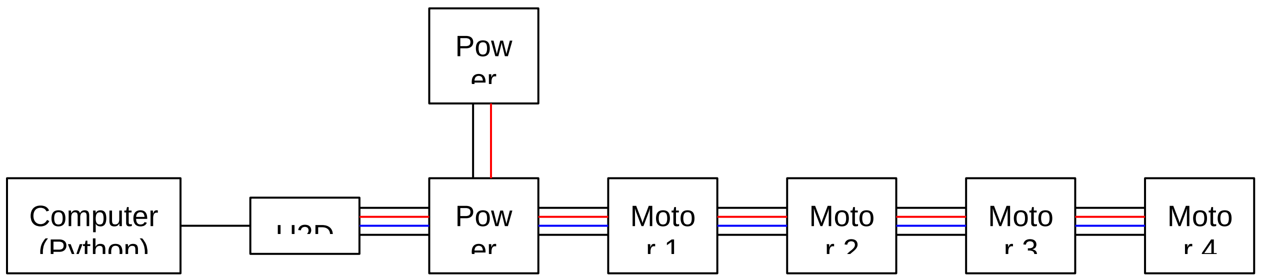
```
        return q

    # Defining function to get current joint space velocity
    def getdQ(self):
        (cur_vel_control_add, cur_vel_len,) =
self.motors[0].CONTROL_TABLE["Present_Velocity"]
        joint_vel_ticks = group_sync_read(self.dxl_io, self.dxl_ids,
cur_vel_control_add, cur_vel_len,)
        dq = np.empty((len(self.motors),), dtype=np.double)
        for i, motor in enumerate(self.motors):
            dq[i] =
self.convert_velocity_tick_to_deg_per_s(joint_vel_ticks[motor.dxl_id])
        return dq

    # Defining function to send PWM signals
    def sendPWM(self, pwm_commands: NDArray[np.int64]):
        goal_pwm_add, goal_pwm_len =
self.motors[0].CONTROL_TABLE["Goal_PWM"]
        pwm = {}
        for i in range(len(self.motors)):
            pwm[self.dxl_ids[i]] = pwm_commands[i]
        group_sync_write(self.dxl_io, pwm, goal_pwm_add, goal_pwm_len,)

    # Defining function to calculate forward kinematics
    def FK(self, q_cur):
        # Accounting for offsets and converting to radians
        q_cur = np.deg2rad(q_cur)
        offset = np.deg2rad([89, 188, 186, 180])
        q_cur = q_cur - offset
        # Accounting for end effector shape
        a = 70
        b = 125 * np.cos(q_cur[3])
        L3 = np.sqrt(a**2 + b**2)
        phi = np.arctan(b/a)
        # Calculating position and orientation
        x = self.link_lengths[0] * np.cos(q_cur[0]) + self.link_lengths[1]
* np.cos(q_cur[0] + q_cur[1]) + L3 * np.cos(q_cur[0] + q_cur[1] + q_cur[2]
+ phi)
```

```
        y = self.link_lengths[0] * np.sin(q_cur[0]) + self.link_lengths[1]
* np.sin(q_cur[0] + q_cur[1]) + L3 * np.sin(q_cur[0] + q_cur[1] + q_cur[2]
+ phi)
        psi = np.rad2deg(q_cur[0] + q_cur[1] + q_cur[2] + np.pi/2)
        # Returning vector of current pose in operational space
        x_cur = [x, y, psi, np.rad2deg(q_cur[3])]
        return x_cur

    # Defining function to calculate Analytical Jacobian
    def J_A(self, q_cur):
        # Accounting for offsets and converting to radians
        q_cur = np.deg2rad(q_cur)
        offset = np.deg2rad([89, 188, 186, 180])
        q_cur = q_cur - offset
        # Accounting for end effector shape
        a = 70
        b = 125 * np.cos(q_cur[3])
        L3 = np.sqrt(a**2 + b**2)
        phi = np.arctan(b/a)
        # Filling each index of Jacobian matrix
        J_A = np.zeros([4, 4])
        J_A[0, 0] = -self.link_lengths[0]*np.sin(q_cur[0]) -
self.link_lengths[1]*np.sin(q_cur[0] + q_cur[1]) - L3 * np.sin(q_cur[0] +
q_cur[1] + q_cur[2] + phi)
        J_A[0, 1] = -self.link_lengths[1] * np.sin(q_cur[0] + q_cur[1]) -
L3 * np.sin(q_cur[0] + q_cur[1] + q_cur[2] + phi)
        J_A[0, 2] = - L3 * np.sin(q_cur[0] + q_cur[1] + q_cur[2] + phi)
        J_A[1, 0] = self.link_lengths[0] * np.cos(q_cur[0]) +
self.link_lengths[1] * np.cos(q_cur[0] + q_cur[1]) + L3 * np.cos(q_cur[0]
+ q_cur[1] + q_cur[2] + phi)
        J_A[1, 1] = self.link_lengths[1] * np.cos(q_cur[0] + q_cur[1]) +
L3 * np.cos(q_cur[0] + q_cur[1] + q_cur[2] + phi)
        J_A[1, 2] = L3 * np.cos(q_cur[0] + q_cur[1] + q_cur[2] + phi)
        J_A[2, 0] = J_A[2, 1] = J_A[2, 2] = J_A[3, 3] = 1
        return J_A

    # Defining function to convert encoder readings to degrees
    @staticmethod
    def convert_encoder_tick_to_deg(motor: DynamixelMotor, pos_tick: int):
        pos_tick = np.asarray(pos_tick).astype(np.int16)
        if pos_tick > 65536:
            pos_tick = -np.invert(pos_tick)
        ang = motor.max_angle * (pos_tick - motor.min_position) /
(motor.max_position + 1 - motor.min_position)
        return ang

    # Defining function to convert encoder readings to velocity
    @staticmethod
    def convert_velocity_tick_to_deg_per_s(velocity: int):
        vel_tick = np.asarray(velocity).astype(np.int16)
        if vel_tick > 65536:
            vel_tick = -np.invert(vel_tick)
        rpm = 6 * (float(vel_tick) * 0.229)
        return rpm

    # Defining function to throw error if Dynamixel connection fails
    def signal_handler(self, *_):
        self.torqueOff()


# Defining main function
if __name__ == "__main__":
    ########## Creating Controller ##########
    # Defining name of COM port to U2D2
    serial_port_name = "COM4"

    # Defining motor IDs
    dxl_ids = 1, 2, 3, 4

    # Defining gain matrices
    K_P = np.array([260e-7, 180e-7, 180e-5, 23.5e-3]) #np.array([5e-5,
19e-6, 29e-4, 3e-2])
    K_I = np.array([90e-8, 100e-8, 180e-5, 400e-4]) #np.array([9e-6, 90e-
7, 40e-4, 55e-3])
    K_D = np.array([40e-10, 190e-10, 170e-7, 35e-5]) #np.array([4e-8, 21e-
9, 29e-7, 16e-4])

    # Defining Controller
    c = PDController(
        serial_port_name=serial_port_name,
        K_P=K_P,
        K_I=K_I,
        K_D=K_D,
        dxl_ids=dxl_ids,
    )


    ########## Trajectory Generation ##########
    # Defining intermediate positions
    home = [-50, 185, 215, 40]
    home_via = [-150, 25, 270, 40]
    cut1_via = [-175, 15, 270, 20]
    cut1 = [-175, 60, 270, -10]
    cut1_post = [-150, 70, 270, 30]
    cut2_via = [-150, 15, 270, 20]
    cut2 = [-150, 60, 270, -10]
    cut2_post = [-150, 70, 270, 30]
    cut3_via = [-125, 15, 270, 20]
    cut3 = [-125, 60, 270, -10]
    cut3_post = [-125, 70, 270, 30]
    move_via = [-100, 80, 270, 30]
    move_start = [-100, 53, 270, -5]
    move_end = [-220, 53, 270, -5]

    n = 19
    # Creating trajectory
    t1, x1, dx1, ddx1 = getTraj(0, 3, 3*n, home, home_via)
    t2, x2, dx2, ddx2 = getTraj(0, 1, 1*n, home_via, cut1_via)
    t3, x3, dx3, ddx3 = getTraj_cutting(0, 3, 0.5, 3*n, cut1_via, cut1)
    t4, x4, dx4, ddx4 = getTraj(0, 1, 1*n, cut1, cut1_post)
    t5, x5, dx5, ddx5 = getTraj(0, 1, 1*n, cut1_post, cut2_via)
    t6, x6, dx6, ddx6 = getTraj_cutting(0, 3, 0.5, 3*n, cut2_via, cut2)
    t7, x7, dx7, ddx7 = getTraj(0, 1, 1*n, cut2, cut2_post)
    t8, x8, dx8, ddx8 = getTraj(0, 1, 1*n, cut2_post, cut3_via)
    t9, x9, dx9, ddx9 = getTraj_cutting(0, 3, 0.5, 3*n, cut3_via, cut3)
    t10, x10, dx10, ddx10 = getTraj(0, 1, 1*n, cut3, cut3_post)
    t11, x11, dx11, ddx11 = getTraj(0, 1, 1*n, cut3_post, move_via)
    t12, x12, dx12, ddx12 = getTraj(0, 2, 2*n, move_via, move_start)
    t13, x13, dx13, ddx13 = getTraj(0, 3, 3*n, move_start, move_end)
    t14, x14, dx14, ddx14 = getTraj(0, 2, 2*n, move_end, home_via)
```

```
    t15, x15, dx15, ddx15 = getTraj(0, 3, 3*n, home_via, home)

    # Storing trajectories together
    times = [t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12, t13, t14,
t15]
    trajectories = [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12,
x13, x14, x15]
    d_trajectories = [dx1, dx2, dx3, dx4, dx5, dx6, dx7, dx8, dx9, dx10,
dx11, dx12, dx13, dx14, dx15]
    dd_trajectories = [ddx1, ddx2, ddx3, ddx4, ddx5, ddx6, ddx7, ddx8,
ddx9, ddx10, ddx11, ddx12, ddx13, ddx14, ddx15]


    ########## Tracking Trajectory ##########
    # Starting motors, running, and stopping
    c.torqueOn()
    t_prev = 0
    for i in range(0, len(times)):
        if i > 0:
            t_prev += times[i-1][-1]
        c.track(times[i], trajectories[i], d_trajectories[i],
dd_trajectories[i], t_prev)
    c.torqueOff()


    ########## Printing Results ##########
    # Saving output
    tim = np.asarray(c.timestamps)
    pos = np.asarray(c.position_history).T
    loc = np.asarray(c.location_history).T

    # Plotting joint space trajectories
    fig1, axs1 = plt.subplots(len(c.motors))
    for i in range(len(c.motors)):
        axs1[i].plot(tim, pos[i])
        axs1[i].set_ylabel('Motor ' +  str(i+1) + ' Angle (deg)',
fontsize=8)
    plt.xlabel("Time (s)")

    # Plotting operational space trajectories
    labels = ['X', 'Y', 'Phi', 'Psi']
    units = ['mm', 'mm', 'deg', 'deg']
    fig2, axs2 = plt.subplots(len(c.motors))
    for i in range(len(c.motors)):
        axs2[i].plot(tim, loc[i])
        time_add = 0
        for j in range(0, len(times)):
            if j > 0:
                time_add += times[j-1][-1]
            axs2[i].plot(times[j][0::1] + time_add, trajectories[j][i],
linestyle='--', color='green')
        axs2[i].set_ylabel(labels[i] + ' Position (' + units[i] + ')',
fontsize=8)
    plt.xlabel("Time (s)")
    plt.suptitle("Operational Space Trajectories")

    # Plotting overhead view
    plt.figure(3)
    plt.plot(loc[0][0], loc[1][0], marker='o', color='red')
    plt.plot(loc[0][-1], loc[1][-1], marker='x', color='red')
    plt.plot(loc[0], loc[1])
    for j in range(0, len(times)):
        plt.plot(trajectories[j][0], trajectories[j][1], linestyle='--',
color='green')
    plt.title("Overhead View")
    plt.xlabel("X Position (mm)")
    plt.ylabel("Y Position (mm)")
    plt.axis('equal')
    plt.show()
```

8.5 Wiring

The following diagram shows the way in which each motor was chained together to receive power from a wall outlet and communication from the U2D2, as well as how the U2D2 serial interpreter was used to relay information to and from a python code on the central computer.


Pow
er
Computer
(Python)
U2D
Pow
er
Moto
r 1
Moto
r 2
Moto
r 3
Moto
r 4