University of British Columbia

Faculty of Applied Sciences

ENGR 499 - Automated Packing Using Collaborative Robots

Final Design Report

Group #62

Ben Ellis	19421809
Ethan Gibbard	81326514
Zach Kelly	41637836
Kade Ronneberg	72729940
Qianyi Wang	21683289

Date: April 15, 2022

Executive Summary

Throughout the world, there are countless production systems that produce products that end up in the packaging stage. The packaging can be for heavy items like cars or smaller such as a CPU for a computer, in any case, the respective care needs to be taken to package each. In the beginning, there was only manual labour where workers would handle and manage the product by hand and package it to specifications. This method leads to many hours of manual repetition which is hard on the worker, human error leading to potential revenue loss, and health and safety risks. Therefore, the need to reduce strain on the worker, boost revenue, and increase safety is in higher demand as technology progresses and allows automation using machines, or robots. The use of robots is very effective and its demand increases every year and is becoming the go-to for most packaging processes.

Building onto the existing Festo MPS production line at UBC Okanagan (UBCO) we created a dual-robot packaging system that packages the small workpieces. The system we developed utilizes two Kinova Gen3Lite robots that can pipeline the packaging as each robot passes the previous task onwards. One robot takes one smaller flatpack cardboard box out of the dispenser and then proceeds to fold the bottom using a complex set of linear and rotational movements. The packaging works under ideal conditions as our solution is hard-coded making everything must be in place otherwise the system fails.

The stakeholder of our project is UBCO where the aim is to incorporate our system into current manufacturing courses to complete the Festo production lineup.

Table of Contents

List of Figures

1.0 Problem Specification

1.1 Background information

An automated packaging process means that products are packaged without manual intervention by humans. With the continuous development, improvement and innovation of technology, automation packaging systems have evolved from singular machines that automate one step in the packaging process to systems that integrate all steps seamlessly into the entire packaging system [1]. Automatic packaging lines will definitely replace traditional packaging methods and labour and become an indispensable part of many production enterprises. Many large enterprises have achieved full automation and no longer use traditional manual packaging [1]. Even small businesses can see the gradual development of packaging mechanization, replacing labour with automatic sealing machines, automatic unpacking machines, carton packing machines, and so on [1]. With the continuous expansion of the packaging market and the increase of choices, automation packaging can reduce labour costs, packaging costs, improve packaging efficiency, and production safety, which has become the consensus of various industries.

Automatic packaging machines come in many forms, but all fall into the automatic or semi-automatic category. Due to the high complexity, variability, and small batches of products, some specialized packaging processes are usually unable to be fully automated by traditional robots. A new mode of manufacturing bots is designed and applied, which is Collaborative Robots (Cobots). The goal of Cobots is to cooperate with humans instead of working in their own space [2]. They are more effective than the larger products of the same kind and have the added benefit of sharing workspaces with humans [2]. In 2012, Cobot was used for research and development, and they proved to be a safer alternative to traditional industrial robots because they are more suitable for operation under dynamic conditions [3]. It is expected that in the next few years, Cobot can effectively replace labour in the industry 4.0 environment such as manufacturing and hospitals [2].

1.2 Problem Statement

Currently, the University of British Columbia - Okanagan (UBCO) requires an automated packaging system for the product from the laboratory-scale FESTO production system by using collaborative robots. This project develops a way to reduce the need for physical labour in a packaging facility while maximizing modularity to align with the concept of cellular manufacturing. The whole packaging process is controlled by two Kinova Gen3Lite robots that implemented Python as the distributed robot control algorithm language. The entire packaging process for the project involves retrieving the product from a laboratory-scale production system, the collaboration between the robots to complete the packaging task, and finally dispensing the packaged product.

2.0 Needs and Constraints Identification

2.1 Stakeholders

UBCO is the primary stakeholder in this project who wants an add-on to their Festo Modular Production System (MPS) lineup which is used for Manufacturing Engineering courses to primarily target learning about Programming Language Controller (PLC). The collaborative robot design would be the final addition to the assembly line where after a workpiece goes through, it can be picked up, packaged, and deposited using two collaborative Kinova robot arms. The design team is enrolled in the university's School of Engineering (SoE) and is a major stakeholder as the success of the project improves performance in the course.

2.2 Needs and Constraints

The packaging process requires locating the workpiece, moving to the location of the workpiece with minimal error (mm), picking up the workpiece (kg), folding a box, placing the workpiece in the box with minimal error(mm), and finally placing the packaged workpiece at the drop off location. Except for provided workpieces and two Kinova robots, this project needs the 2" x 2" x 2" packaging cardboard boxes. In order to help fold the box, some mechanisms were designed and 3D printed. A simulation by CIROS was made to simulate the motion of two robotic arms and the availability of the packaging process.

There are a few mechanical limitations that serve as constraints for this project. The Kinova Gen3Lite robot has 6 joints, and each has a rotational movement limit of 300 degrees. It makes determining the joint coordinates of the robot arm becomes more complex and time-consuming. Because of the programming to the robots' coordinates, the entire packaging process needs high precision. The location change of mechanisms and workpieces may cause the packaging process to be false. In addition, due to the lack of a computer vision system, the system cannot self-check its packaging process, and it is impossible to flexibly detect and grab the workpiece at any position. Furthermore, the timeline of this project is a constraint as there is a period of eight months from September 2021 to April 2022 to complete the project.

2.3 Economic and Safety Consideration

The budget for this project to be done, as the Kinova robots are already supplied, is $300. Safety is the next big concern as the robot may move in ways that may endanger students or staff standing too close to the robot. The robot would not cause harm as it does not move with an incredible amount of force but we will draw outlines around the workstation so that if the robot arms are being used people should be outside of the range of the arms. The SolidWorks model for the ranges can be seen below in Figure 6 which follows a radius around the robot that's approximately 45 inches each; the intersection between the two hemispheres is where the two robots could collide. This collision volume is planned to be known such that our code makes sure the arms do not collide.

3.0 Design Process

The design process section describes what we designed, and how our system works. It describes in detail all the parts needed to complete the automated task, and why they are needed.

3.1 Packaging

3.1.0 Cardboard box

The packaging system is controlled by the two Kinova Gen3Lite robots. They are placed at the opposite corners of the table for completing the packaging work. The process begins with picking up a single cardboard box from a dispenser, folding the base of the box, placing the product inside the box, folding the top of the box, and then moving the package to a final location. The box we decided to use for this project has a self-locking design, eliminating the need to tape the bottom or top of the box to secure the package. However, the gripper of Kinova Gen3Lite is not flexible enough and has large inaccuracies in the packaging processes of some small-size products. For example, the cardboard box used in this project is 2” x 2” x 2”, and is difficult to fold using only the two arms. The box, its flat pack, and how it folds together are in Figure 1 below. The robots will have to assemble the box according to the procedure on the left of Figure 1. Note that the last bottom flap is not shown as being folded, it is done by pushing it into the bottom until it locks behind the other parts.

Figure 1. The 2x2x2 box, the procedure to fold, and flat-pack [4].

3.1.1 Platform

The Kinova Gen3 Lite robots have a very large range of motion, however, when trying to manipulate objects at the same height as the base of the robot, we came across issues. These issues come due to the first joint of the robot being perpendicular to the base, so when trying to work with objects at the base height, the range of motion for the other joints is severely decreased. This resulted in singularities, where the robot would reach the end of its range of motion and refuse to continue moving in specific directions. To compensate for this issue, we decided to raise the operating level with respect to the base of the robots, approximately to the height of the robot's first joint to maximize its range of motion. We did this just by attaching all of our workspace moulds to a box of the chosen height and placing the box on the table. This worked to eliminate the majority of the singularities we were experiencing and allowed us to proceed with the packaging process.

3.1.2 Storage and dispensing unit

The boxes are initially in flattened form, which allowed us to pack a lot of them into a small amount of space. We designed a holder cartridge (as shown in figure 2) to hold 50 boxes at a time. The cartridge needed the capabilities to allow a single box to be grabbed at a time by the robot, without grabbing any of the other boxes. We designed a dispensing conveyor belt that goes underneath the holder cartridge and attached a position sensor for feedback capabilities. The conveyor belt was made out of a stepper motor, some bearings, and a thick elastic band, and was controlled by an Arduino. The holder cartridge now had the capabilities to dispense a single box at a time. When the robot would pick up a dispensed box, the position sensor would send a signal to the Arduino to dispense a new box, turning on the conveyor belt.

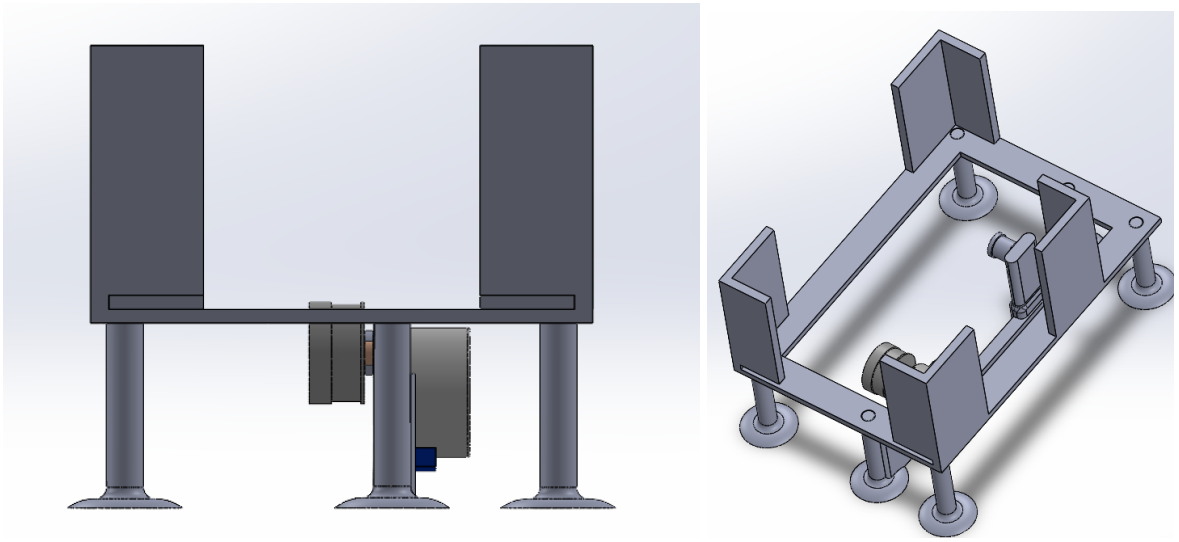

Figure 2. Flatpack storage and dispenser, stepper motor assembly

3.1.3 Re-gripper post

After picking up the cardboard box from the holder cartridge, The Kinova robot would apply force to the sides of the box. This would open up the box, and change it from its flatpack state to its opened state. The next step of the design was to reposition the arm so that instead of holding the box diagonally, it readjust its grip so that it was holding the box parallel to the folds. This would allow a much more stable grip on the box and would help with folding the bottom of the box in the future. We designed a regripping apparatus, which was 3D printed (as seen in figure 3). The regripper functions by having the box placed on top of it, holding it in place while the robot adjusted its grip to be parallel. The regripper was fixed to the table, and allowed for consistent grip position of the box, allowing the box to be in the exact same position in the robot's gripper every time.

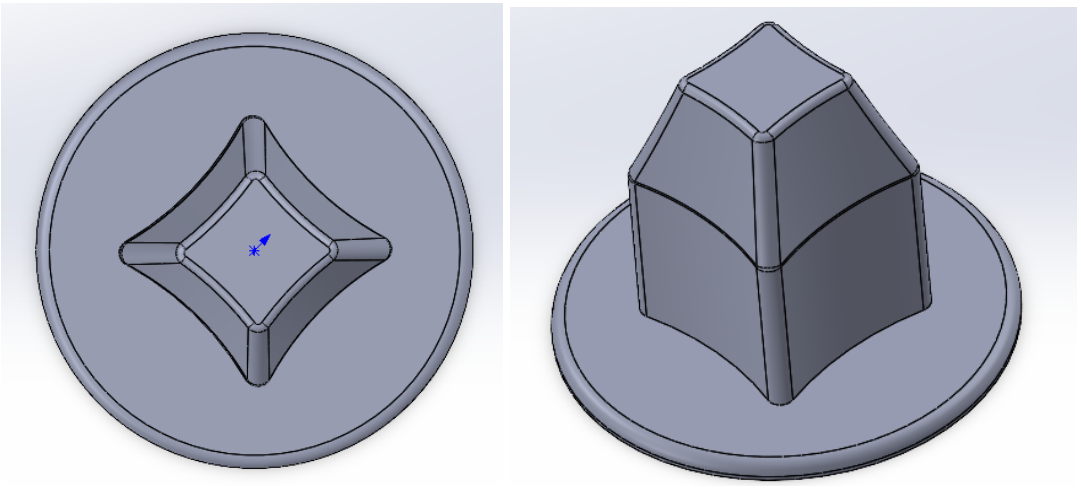

Figure 3. The mechanism for regripping the box

3.1.4 Fin Mechanism

Due to the intricate design of the box, we found folding the bottom just using the robots to be difficult. Therefore, a mechanism was designed to help fold the cardboard box (as shown in Figure 4). It is fixed to the table and is composed of four curved prismatic columns and a base. After squeezing the flat-packed cardboard into a 3D box, the robotic arm can move the box in a horizontal direction to one of the fins on the mechanism, to fold the first bottom side. The arm then moves the box to the right to fold the first side flap. Then the box can be rotated 90 degrees for folding the second side flap. There is enough distance between each fin to ensure that the unfolded sides will not be touched when the box is rotated. Because of the self-locking design, after folding the last side, the robotic arm can position the cardboard box on top of one of the fins, and then move it vertically downwards to lock the flaps of the bottom side.

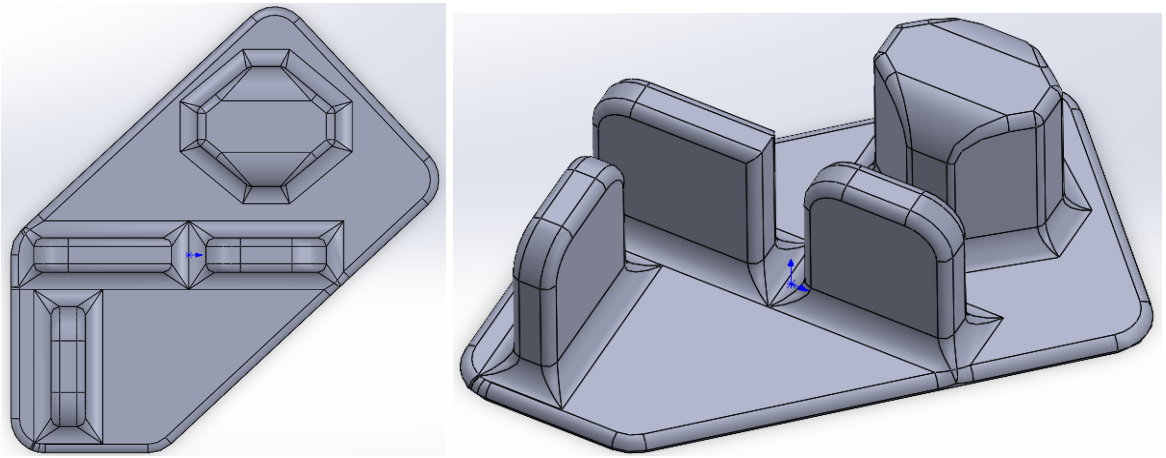

Figure 4. The mechanism for folding the cardboard box.

3.1.5 Box mold

After folding the bottom of the box, the first arm moves the partially folded box to a position accessible by the second robot. Due to some sway in the robots, while trying to be in a steady position, we decided to create a mold to hold the folded box while the second robot would fold the top of the box. This also allowed the first robot to let go of the box, and go back to the beginning of the process to start folding a new box, while the second robot is finishing the packaging of the product. The repositioning mold we created was 3D printed as well, as seen in Figure 5. The mold was a decently tight fit to the box, so it would take some force to put the box in the mold, but after it was placed, it would not move during the rest of the packaging process until the second arm was ready to remove it. After placing the box in the mold, the second robot then picks up the product to be packaged, in our case, a metallic cylinder from the Festo Didactic MPS after having the lid placed. The arm proceeds to move the cylinder overtop of the box, places it inside the box, and then folds the top of the box. After finishing folding the top of the box, the arm then moves the packaged product to its final destination, having completed the full process. This whole process then repeats continuously, packaging products non-stop.

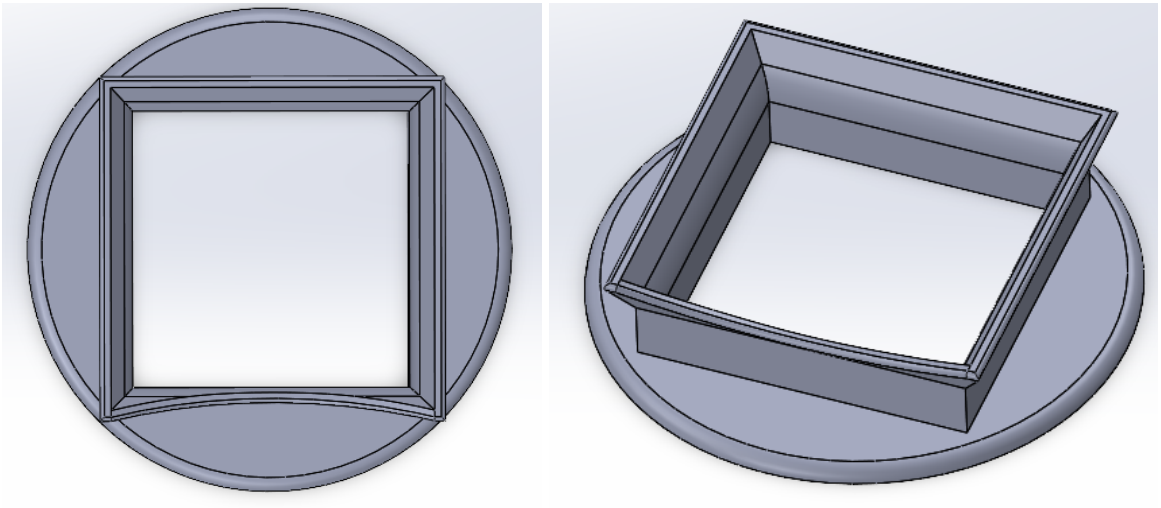

Figure 5. The mechanism for holding the cardboard box

3.2 Synchronization

In the entire procedure, the connection and the cooperation between the two robots is one of the most challenging tasks that need to be perfected. The two Kinova robots are completely independent of each other. They have different IP addresses and they are programmed separately. The program must communicate with both robots, to coordinate tasks. It must be able to let each robot know what step of the task it is on, and give feedback when a task has been completed and the next task needs to begin. It is not acceptable if the two robotic arms start their tasks at the same time in the packaging station since some steps cannot be started until the previous step is completed. For example, the first robot has the task to open the flat-packed cardboard into a 3D box and encapsulate the bottom. Therefore, the second robot cannot pick up the puck and put it in the box until the bottom of the box is folded.

There initially were issues trying to connect the robots both to the same script from a single controller. If both robots were connected to a single computer using the micro-USB cords provided, they both showed up on the computer with the same IP address, so the computer did not know how to differentiate between the robots. We were able to find a workaround for this issue by connecting the second arm via an ethernet port. This allowed the second robot to have a slightly different IP address, allowing the controller to differentiate between robot 1, and robot 2, sending each robot its own specific tasks. We were able to control both robots synchronously using a parallel script, using one core for each robot. We then synchronized

their movements using locks and shared memory storing the current step each arm was working on at any given time.

3.3 Feedback

This project lacks an inspection system. The principle of the robotic arms' motion is treating the package station as a plane and defining the operating space spatially using 3D coordinates and frames. Therefore, the locations are known and the gripper of the arm can move to a specific position and grab an object. However, if subject to changes in the testing environment, such as slight displacement of the 3D printed molds with reference to the Kinovas' base, or rotation of the base of the Kinovas, the packaging process is misaligned. The Kinovas still navigate through the hardcoded positions, meaning the packaging process is not repeated as previously intended. This may result in failure to grab the box, incorrect folding of the bottom, or other failures. To fix these issues, positional feedback is needed to update the misaligned trajectory of the Kinovas'. Computer vision is a potential approach to address the navigation problem.

Another problem is when folding the box, the gripper will catch both sides of the cardboard and apply a force to squeeze it, changing it from a flat-pack to a 3D box. This process is also difficult to achieve because when an unfolded box has applied pressure, rather than opening up, it would tend to just fold in half. To compensate for this, we managed to incorporate rotational motion into the initial unfolding operation, however, this still lacks consistency. An inspection system is necessary after each process to check if the process is working or has some errors. If it is detected that a step has failed, the robots can work to fix the situation, and then proceed to the next step.

4.0 Experimentation

The experimentation section describes the problems encountered in the design process and how they lead to our current solutions. Our current solutions were obtained through iterative development and testing of designs, with the focus on better addressing the corresponding problem.

4.1 Physical Design Components

The physical components used to assist the packaging process include the fin mechanism, the storage and dispensing unit, the re-gripper post, and the box mold.

The fin mechanism, as seen in figure 4, was introduced early on in the project, to propose a better solution to folding the bottom flaps of the box, rather than only using the gripper attachments of the Kinova robots. Introducing some form of assistance in the process was a necessary design step, hence there are four flaps to fold and only two contact points on the Kinova attachments. The design concept carried one coupled problem, which is to determine the optimal sequence of rotations and translations of the box and the orientation and spacing of the fins. Numerous iterations of the fin mechanism were printed and tested by replicating that stage of the packaging process to maximize simplicity and consistency.

With the desire to maximize automation, the storage and dispensing unit, as shown in figure 2, was introduced to provide automatic input to the Kinova robots. This required the flat packs to be dispensed a specified distance out. This requirement had to consider the weight of the stack, the friction produced between the bottom flat pack and the available motor torque that can be produced from the stepper motor. The process was optimized via experimentation of different applied weights and areas of contact between the flat pack and propulsion system. The reflectivity sensor is needed to provide instruction to the stepper motor to start dispensing, which adds to the complexity of the problem. The reflectivity sensor could consistently obtain accurate sensor data, however, is dependent on the proper function of the motor-conveyor system to produce automation.

The re-gripper post, as seen in figure 3, provides a solution to map between the dispensing unit and the fin mechanism. The problem relies on the fact that the flat pack must be initially

opened via pinching the opposite edges and then repositioned to the side faces of the box prior to the fin mechanism. The solution must allow the current Kinova to release the hold of the box to a static location that kept the box open, to where it could come back and re-grip to the desired location intended for the next stage, the fin mechanism. The design adopted a chamfer at the top of the structure to accommodate localization error and a tall enough base to support the forces exerted when re-gripped.

The box mold, as shown in figure 5, was last to be developed to assist the automation process. The need arose from the challenges experienced in folding the top flap of the box. There were three problems with the pre-existing method, the placement of the box from the first Kinova could not always be perfectly located with respect to the second Kinova, and the first struggled to not move when subject to and not subject to forces from the second. The design could remove localization errors due to the extruded outward draft added to the mold, meaning that if the placement of the box is misaligned from the Kinova the extrudes could re-direct the box into alignment. This design also reduced the number of dependencies of the second Kinova, hence the Kinova now only depends on the position of the box mold relative to its prescribed coordinate list.

4.3 Robotic Manipulation and Control

The heart of the automation process attributes to the Kinova Gen3Lite robots and the program which controls them. In an attempt to automate the packaging process, there arose problems associated with the manipulation and control. The manipulation problem is concerned with the available positions and orientations of the gripper (end-effector), whereas the control problem is concerned with the Kinovas' lack of knowledge of its environment. The dextrous workspace or the set of points the end-effector can reach in any orientation is illustrated in the figure below.

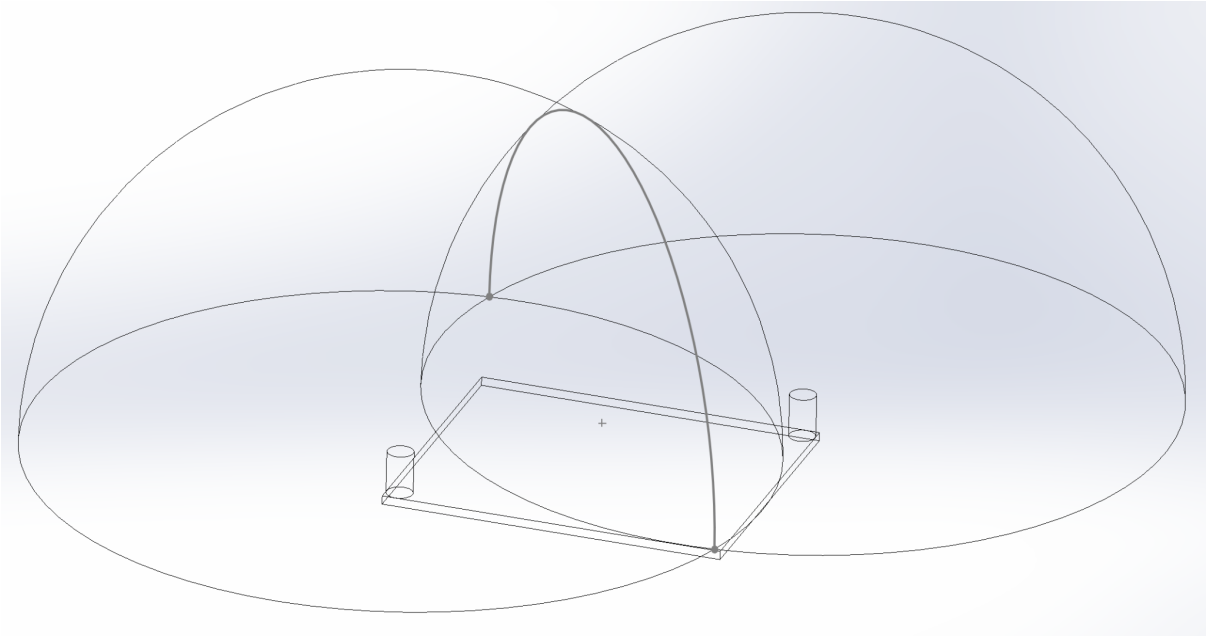

Figure 6. Approximate representation of the dextrous workspace for Kinova Gen3Lite

The Kinova Gen3Lite is a seven degree of freedom robotic manipulator, however, not all joint configurations are possible within their respective semi-spheres. Experimentation revealed that the manipulators frequently failed to reach the desired positions and orientations within the figure 6 workspace when packaging at the base level. However, the number of unreachable positions and orientations was substantially reduced when packaging was raised to joint one level.

Provided the end-effector is able to reach the set of recorded coordinates the packaging process can be completed assuming the positions of all physical components still align with the recordings. However, if subject to changes in the environment the control program will send the Kinovas to the incorrect location, resulting in a failure. The control program fails to provide the Kinova with successful direction because the current program only attempts to reproduce the pre-recorded coordinate list, which is given as an input. Therefore to achieve robustness of the packaging process, the program needs to update its understanding of the environment to correct for disturbances. A machine learning development along with exteroceptive sensors to obtain feature data of the environment may be considered as a solution to correct for disturbances.

5.0 Final Design

The packaging process begins at the 3D printed flatpack storage and dispensing unit (Figure 7). The storage unit is filled with flat-pack cardboard boxes over a stepper motor. When the reflectivity sensor located just outside of the storage is off the stepper motor will start dispensing a cardboard box into position for the first Gen3Lite arm to grip and start the assembly process. The stepper motor is hooked up much like a treadmill and when running it uses friction to grip the box as it rolls it through the slit as seen in Figure 7. The stepper motor is controlled by a raspberry pi located right next to the storage and dispenser unit to reduce processing time and requires very little energy. The stepper motor and treadmill design were chosen due to their incredibly low cost however the treadmill design does require pressure to be inserted on the flat-pack boxes to ensure traction.

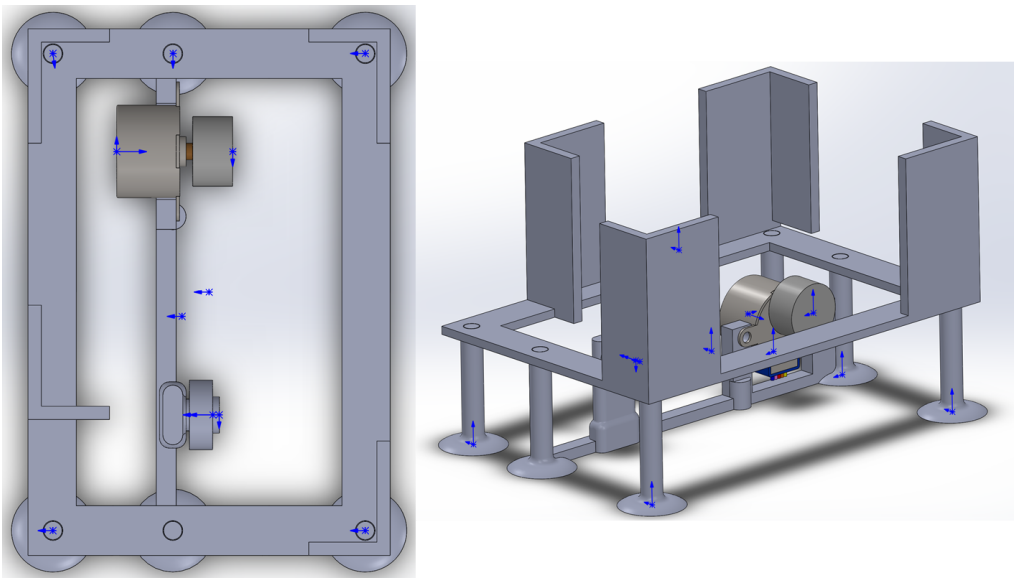

Figure 7. Flatpack Dispenser

The robotic arm swings into the gripping position allocated by the first step in the python waypoint program. The python waypoint program is from Kinova's GitHub page and allows users to program each robotic joint at a desired angle. In trials, we manually moved the robotic arm to the desired position using an Xbox controller, recorded all coordinates and input them into an excel spreadsheet. From the excel spreadsheet, we were able to compile a program that went through every waypoint completing the entirety of the folding process. The arms are controlled using a parallel script, as shown in figure 8, allowing each robot to be synchronously controlled. Each robot is controlled by a position list, telling it what to do and when sequentially. The position list is created by compiling an array of arrays of 9 variables, as seen in figure 9. Each sub array corresponds with its own position. The first 6

variables are doubles which represent the joint angles for that specific position, going from joint 1 to joint 6 in order. The 7th decimal is the gripper position, 0 being fully opened, 1 being fully closed, and any decimal in between being a fraction of the opened position. The 8th variable is the step required to execute that position, and the 9th variable is the step to progress to after executing that position. The steps are a method of communicating between the arms. The code determining whether to proceed with a position or not can be seen in figure 10. Before executing any position, the program will check if both arms are in the correct step to proceed with that position. After executing any position, the program will check if it should change the step number for one of the arms, and change it if necessary. If one of the arms is not in the step required to execute a position for the other arm, the other arm will wait until the first robot has progressed to the required step, allowing for synchronization between the robots. This ensures that no robot will try to execute a position while the other robot is in a potential collision position, eliminating any chance of collisions between the robots.

```
def main():

    # Import the utilities helper module
    sys.path.insert(0, os.path.join(os.path.dirname(__file__), ".."))
    stage = Value('i', 1)
    next_stage_1 = Value('i', 2)
    next_stage_2 = Value('i', 2)
    lockCurrent = Lock()
    lockNext = Lock()
    a1 = Process(target=arm1, args = (stage,next_stage_1, next_stage_2, lockNext))
    a2 = Process(target=arm2, args = (stage,next_stage_1, next_stage_2, lockNext))
    a1.start()
    a2.start()
    a1.join()
    a2.join()
```

Figure 8. Parallel control code

```
[339.857,13.981,285.868,267.119,92.039,66.731,0,1,1],
```

Figure 9. Position List example position

```
for a in range(len(posList)):
    while(stage.value!=posList[a][7]):
        lockNext.acquire()
        print(stage.value)
        print(posList[a][7])
        if(next_stage_1.value==next_stage_2.value & next_stage_1.value!=stage.value):
            stage.value = next_stage_1.value;
        lockNext.release()
        time.sleep(0.2)


    success &= angular_waypoints(base, posList[a], router)
    lockNext.acquire()
    next_stage_1.value = posList[a][8]
    lockNext.release()
```

Figure 10. Synchronization between arms code

The robotic arm grips the box edges, shifting it out of the flat-pack storage unit and increasing its grip strength once the box is free to open the box to be ready for folding. This readies the robot to move onto stage two which is placing the newly opened box on the re-gripper column located right next to the flatpack dispenser(Seen in Figure 11). The robot makes two movements, one to place the box on the re-gripper column and another to push down the box to maximize accuracy. This design was implemented to ensure that when moving onto stage three, the folding of the bottom flaps, the gripper would be in an ideal position so that it would not catch and fail to produce a folded box.

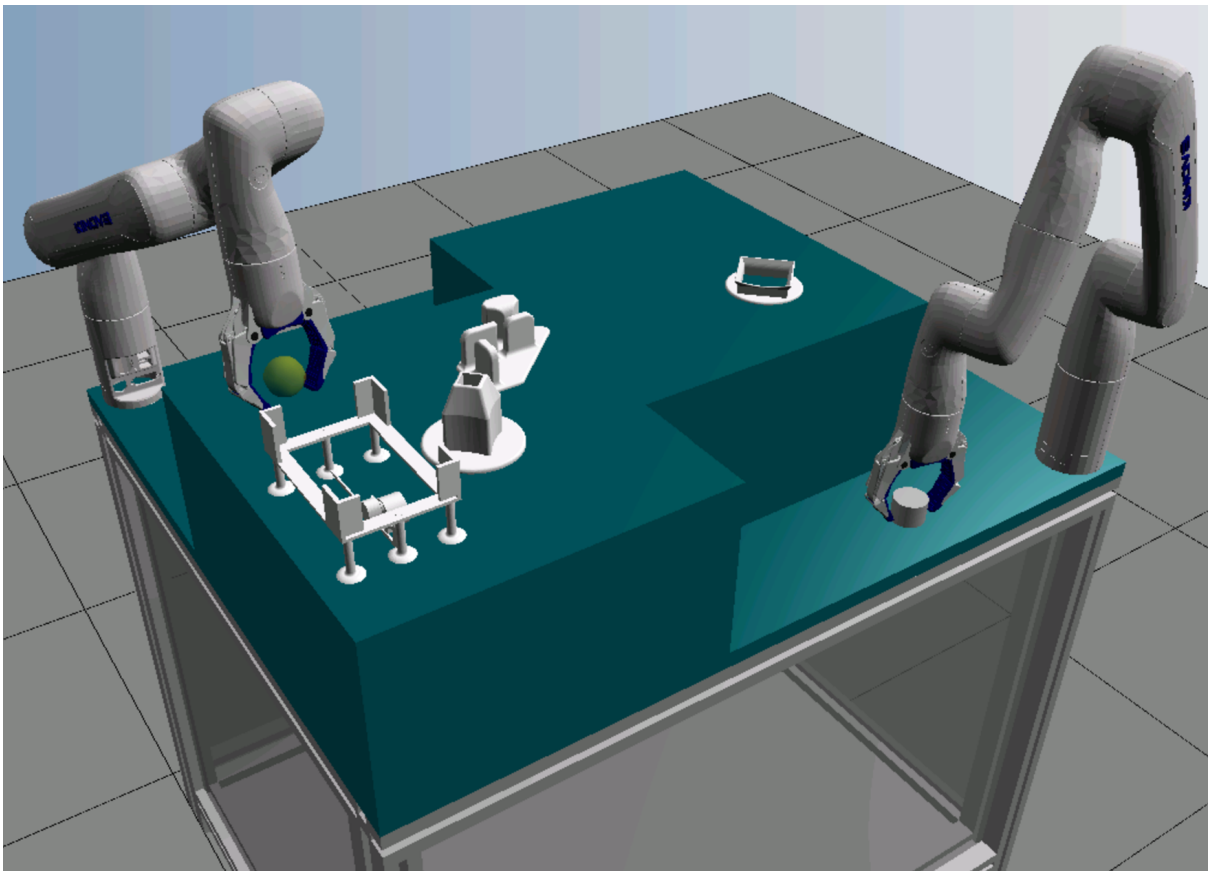

Figure 11. CIROS Simulation Environment.

Stage three is done in two steps, the first is the initial grabbing of the box has to be very precise as grabbing too high or too low will cause critical faults in the folding operation. As seen below in Figure 12 a ROYGB(Red-Orange-Yellow-Green-Blue) order can be seen for exactly how the folding is done. The process starts with two simple folds by dragging the cardboard box over the fins, followed by a ninety-degree rotation, followed by two more folds and finally, the robot pushes down the box to lock it into place. This is possible due to the cardboard boxes chosen for this project with their built-in locking mechanism.

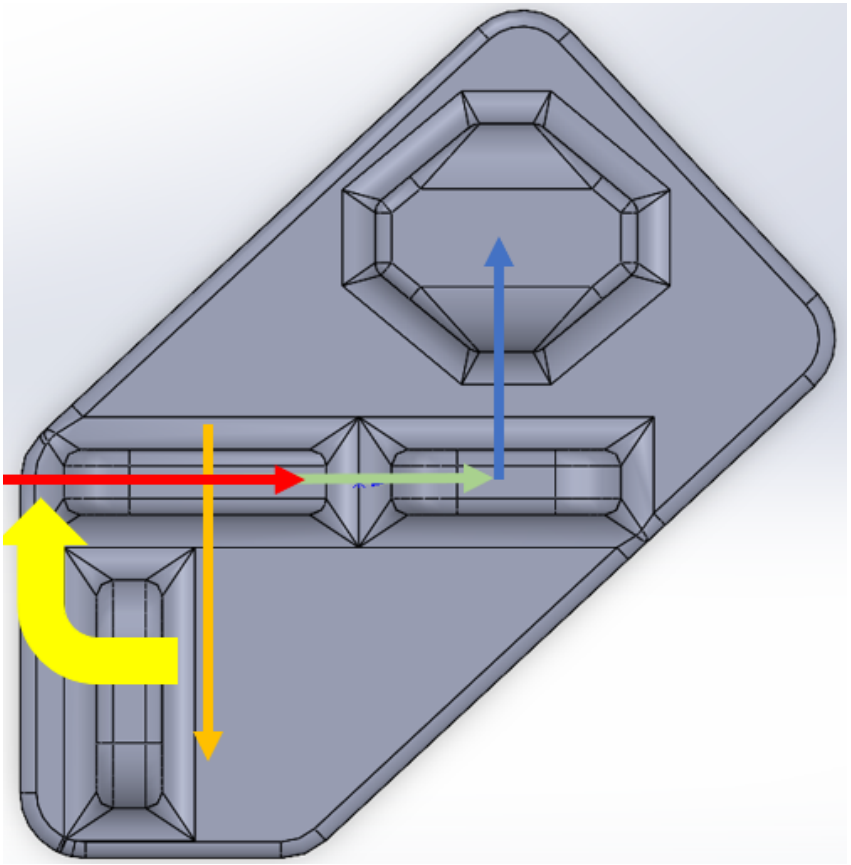

Figure 12. Movement for Folding the Bottom of the Box

Upon completion of the folding task, the robot moves to the drop-off location placing it inside of a 3D printed supporting mechanism to lock it in place and move on to folding the next box while the second Gen3Lite robot finishes the packaging task.

When the first robot arm has placed the box inside of the box supporting mechanism the python code moves on to the next stage giving the second robot arm a green light to finish the packaging. The second robot starts by grabbing the pay-load from the dispensed location and moves to place it inside the box. Once the payload has been dropped the arm moves down to push the payload and box securely into the box supporting mechanism. The predetermined waypoint code proceeds to fold in the 2 side fins and finally the top of the box. Locking the top of the box is a challenging part so the robot first pushes down the top completely before pushing down the locking fold. Once both are aligned with the box it lifts the lid slightly to allow the locking fin to quickly slide in and the fold is complete. The second robot arm taps the two corners to make sure it's locked into place and delivers the folded cardboard off to the next station to be delivered.

6.0 Conclusion

Under the conditions where all assisting mechanisms and Kinovas are positioned in an exact way, our system can package the workpieces into a small cardboard box. While the bottom of the box almost always folds properly, the lid is the most difficult to fold as it requires very fine maneuvers. Most of the difficulties we encountered were troubles with coding as the existing documentation is poor and otherwise non-existent ultimately leading to the use of basic waypoint manipulations. In the future, we would take a new approach altogether as the methods provided non-ideal throughput and reliability. The incorporation of cameras to obtain position data alongside machine learning could remove the need for exact placements and also streamline the process entirely as optimal paths would be found. Our solution provides ample opportunity for students to improve upon our design and take it further with the proposed design improvements mentioned in the sections above and below. The use of robotics in packaging systems increases daily with advancements as they become more cost and time effective. This project showed the successful implementation of an automated packaging process under the ideal conditions.

7.0 Future Work

7.1 Computer vision

As discussed in the design process section, there is no position feedback to the controller to know if any issues have occurred in the process. If any feature in the environment is changed the packaging process is distributed and often results in failure. In order to compensate for disturbances a visual-based localization and mapping method is considered. The Kinova would be equipped with a visual module, such as the cameras of Intel® RealSenseTM Depth Module D400 series. The subsystem components include the stereo depth module, vision processor, and colour image signal processing function connected to the host processor via USB 2.0/USB 3.1 Gen 1 or MIPI1 [5]. Figure 13 shows the stereo depth module of Intel® RealSenseTM Depth Module D410.

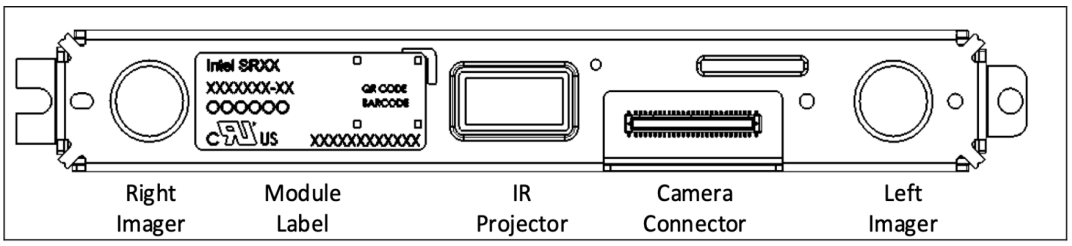

Figure 13. Stereo Depth Module (Intel® RealSense Depth Module D410)

The vision module is to classify the features in the environment and estimate the position of the features relative to Kinovas, using captured RGB images. This method requires major advancements to the software, however, it no longer relies solely on human input, hence it is capable of making corrections based on observational data obtained from the visual module. Therefore the Kinvoas could adapt to disturbances; increasing the overall robustness and consistency of the packaging process.

7.2 Prototype improvements

As mentioned in the design process, the Kinovas' run into fewer singularities when the packaging process is raised to approximately joint 1 level. In the current prototype, the packaging process had been raised by a large cardboard box, which was placed on the same table the Kinovas are mounted to. An improved alternative is to replace the cardboard box with a more rigid and levelled structure. Rigidity is important because with the cardboard box, the molds on top were subject to displacements and curvatures on the surfaces. In addition, replacing the 3D printed components with more sturdy alternatives would help minimize disturbances to the packaging process.

References

[1] Baroro, J. M. M., Alipio, M. I., Huang, M. L. T., Ricamara, T. M., & Beltran Jr, A. A. (2014). “Automation of packaging and material handling using programmable logic controller”. International Journal of Scientific Engineering and Technology, 3(6), 767-770.

[2] L. Rozo, S. Calinon, D. G. Caldwell, P. Jiménez and C. Torras, "Learning Physics Collaborative Robot Behaviors From Human Demonstrations," in IEEE Transactions on Robotics, vol. 32, no. 3, pp. 513-527, June 2016, doi: 10.1109/TRO.2016.2540623.

[3] M. Kwon, E. Biyik, A. Talati, K. Bhasin, D. P. Losey, and D. Sadigh, (2020), “When Humans Aren't Optimal: Robots that Collaborate with Risk-Aware Humans”, In Proceedings of the 2020 ACM/IEEE International Conference on Human-Robot Interaction (HRI '20). Association for Computing Machinery, New York, NY, USA, 43–52.

[4] BENECREATE. (2018, December 13). BENECREAT 60PCS Gift Boxes White Paper Boxes Party Favor Boxes 2 x 2 x 2 Inches with Lids for Gift Wrapping, Wedding Party Favors [web log]. Retrieved November 19, 2021, from https://www.amazon.com/BENECREAT-Inches-Wrapping-Wedding-Favors/dp/B07LCJ96X H

[5] Bulanon, D. M., Burr, C., DeVlieg, M., Braddock, T., & Allen, B. (2021). “Development of a Visual Servo System for Robotic Fruit Harvesting”. AgriEngineering, 3(4), 840–852. doi:10.3390/agriengineering3040053

[6] Ellis, B (2022) Final Control Code (Version 1) [Source code]. https://github.com/bennellis/CapstoneGroup62/blob/main/Final%20Control%20Code.py

[7] Ellis, B (2022) Arduino stepper control (Version 1) [Source code]. https://github.com/bennellis/CapstoneGroup62/blob/main/arduino_stepper_control.ino

[8] Maisonneuve, F (2021) Kinova Example Code (Version 67) [Source Code]. https://github.com/Kinovarobotics/kortex/tree/master/api_python/examples

Robot Control Python Code [6]

```
#! /usr/bin/env python3

###
# KINOVA (R) KORTEX (TM)
#
# Copyright (c) 2018 Kinova inc. All rights reserved.
#
# This software may be modified and distributed
# under the terms of the BSD 3-Clause license.
#
# Refer to the LICENSE file for details.
#
###

import sys
import os
import time



import threading
from multiprocessing import Process, Value, Lock

from kortex_api.autogen.client_stubs.BaseClientRpc import BaseClient
from kortex_api.autogen.client_stubs.BaseCyclicClientRpc import BaseCyclicClient

from kortex_api.autogen.messages import Base_pb2, BaseCyclic_pb2, Common_pb2

# Maximum allowed waiting time during actions (in seconds)
TIMEOUT_DURATION = 20
```

```
# Create closure to set an event after an END or an ABORT
def gripperGoTo(perc, router, base):
    if perc > 1:
        perc = 1
    if perc < 0.01:
        perc = 0.01

    gripper_command = Base_pb2.GripperCommand()
    gripper_request = Base_pb2.GripperRequest()
    finger = gripper_command.gripper.finger.add()



    print("Performing gripper test in position...")
    gripper_command.mode = Base_pb2.GRIPPER_POSITION
    gripper_request.mode = Base_pb2.GRIPPER_POSITION

    position = perc
    finger.finger_identifier = 1
    finger.value = position
    print("Going to position {:0.2f}...".format(perc))
    base.SendGripperCommand(gripper_command)

    startTime = time.time()




    while True:
        gripper_measure = base.GetMeasuredGripperMovement(gripper_request)
        if len (gripper_measure.finger):
            print("Current position is : {0}".format(gripper_measure.finger[0].value))
            dif = gripper_measure.finger[0].value - position
            now = time.time()
            if (dif < 0.01) & (dif > - 0.01):
```

```
                break
            if(now-startTime > 1):
                if(dif < 0.05) & (dif > -0.05):
                    break
        else: # Else, no finger present in answer, end loop
            break
    time.sleep(0.1)



def check_for_end_or_abort(e):
    """Return a closure checking for END or ABORT notifications

    Arguments:
    e -- event to signal when the action is completed
        (will be set when an END or ABORT occurs)
    """
    def check(notification, e = e):
        print("EVENT : " + \
              Base_pb2.ActionEvent.Name(notification.action_event))
        if notification.action_event == Base_pb2.ACTION_END \
        or notification.action_event == Base_pb2.ACTION_ABORT:
            e.set()
    return check

def example_move_to_home_position(base):
    # Make sure the arm is in Single Level Servoing mode
    base_servo_mode = Base_pb2.ServoingModeInformation()
    base_servo_mode.servoing_mode = Base_pb2.SINGLE_LEVEL_SERVOING
    base.SetServoingMode(base_servo_mode)

    # Move arm to ready position
    print("Moving the arm to a safe position")
    action_type = Base_pb2.RequestedActionType()
```

```
    action_type.action_type = Base_pb2.REACH_JOINT_ANGLES
    action_list = base.ReadAllActions(action_type)
    action_handle = None
    for action in action_list.action_list:
        if action.name == "Home":
            action_handle = action.handle

    if action_handle == None:
        print("Can't reach safe position. Exiting")
        return False

    e = threading.Event()
    notification_handle = base.OnNotificationActionTopic(
        check_for_end_or_abort(e),
        Base_pb2.NotificationOptions()
    )

    base.ExecuteActionFromReference(action_handle)
    finished = e.wait(TIMEOUT_DURATION)
    base.Unsubscribe(notification_handle)

    if finished:
        print("Safe position reached")
    else:
        print("Timeout on action notification wait")
    return finished


def angular_waypoints(base, posList, router):
    print("Starting angular action movement ...")
    action = Base_pb2.Action()
    action.name = "Example angular action movement"
    action.application_data = ""
```

```
actuator_count = base.GetActuatorCount()

# go to posList

#

for joint_id in range(actuator_count.count):
    joint_angle = action.reach_joint_angles.joint_angles.joint_angles.add()
    joint_angle.joint_identifier = joint_id
    joint_angle.value = posList[joint_id]



e = threading.Event()
notification_handle = base.OnNotificationActionTopic(
    check_for_end_or_abort(e),
    Base_pb2.NotificationOptions()
)

print("Executing action")
base.ExecuteAction(action)


print("Waiting for movement to finish ...")
finished = e.wait(TIMEOUT_DURATION)
base.Unsubscribe(notification_handle)

if finished:
    print("Angular movement completed")
    if(posList[6] != -1):
        gripperGoTo(posList[6], router, base)
else:
    print("Timeout on action notification wait")
return finished
```

```
def arm1(stage,next_stage_1, next_stage_2, lockNext):

    import argparse
    sys.path.insert(0, os.path.join(os.path.dirname(__file__), ".."))
    import utilities
    # Parse arguments
    args = utilities.parseConnectionArguments()

    # Create connection to the device and get the router
    with utilities.DeviceConnection.createTcpConnection(args) as router:

        # Create required services
        base = BaseClient(router)
        base_cyclic = BaseCyclicClient(router)

        # Example core
        success = True



        success &= example_move_to_home_position(base) #home position

            #input position list for arm 2 below. 7th array index is used to control gripper. 1.0 is fully closed, 0.0 is fully open. if left as -1, will not change gripper. 8th and 9th array indexes are for

            #synchronization between the steps. 8th index is step required by both arms before execution, and 9th index is step to proceed to for that arm after executing that position.
        posList = [ \
[339.857,13.981,285.868,267.119,92.039,66.731,0,1,1],\
[345.411,57.817,351.299,271.991,109.649,73.144,-1,1,1],\
[346.658,53.41,339.26,270.958,102.2,75,0.315,1,1],\
[347.05,51.692,336.811,270.946,101.532,75.249,-1,1,1],\
[346.829,48.242,330.236,270.946,98.444,75,-1,1,1],\
```

[346.704,46.572,327.111,270.949,96.985,74.837,-1,1,1],\
[346.575,45.284,324.792,270.951,95.915,74.751,-1,1,1],\
[346.408,43.196,321.038,270.955,94.25,74.541,-1,1,1],\
[351.488,41.235,320.295,261.335,93.38,78.816,-1,1,1],\
[356.132,41.495,322.294,254.3,95.011,82.443,0.34,1,1],\
[358.773,43.033,324.721,249.352,97.13,84.186,0.4,1,1],\
[4.506,44.685,327.912,238.118,100.224,86.996,-1,1,1],\
[5.012,37.718,315.556,238.408,94.642,90.379,-1,1,1],\
[5.591,27.888,299.696,238.525,89.851,93.922,-1,1,1],\

[3.867,26.318,295.344,261.563,86.376,94.091,0.48,1,1],\
[3.868,26.321,295.307,261.552,357.887,94.088,-1,1,1],\
[6.901,32.696,262.613,272.712,318.824,83.512,-1,1,1],\
[18.406,37.019,263.868,304.597,309.871,61.392,-1,1,1],\
[18.18,41.712,262.517,301.784,304.772,65.74,-1,1,1],\
[18.068,43.623,262.281,300.941,302.985,67.177,-1,1,1],\
[18.062,44.495,262.203,300.647,302.165,67.714,-1,1,1],\
[18.051,45.311,262.156,300.311,301.467,68.285,-1,1,1],\
[18,47.322,262.154,299.651,299.725,69.441,-1,1,1],\
[17.961,47.818,262.154,299.55,299.302,69.766,0.27,1,1],\
[18.059,43.877,262.252,300.877,302.747,67.364,0.492,1,1],\
[18.054,44.53,262.199,300.63,302.136,67.7,-1,1,1],\
[18.046,45.327,262.156,300.306,301.456,68.291,-1,1,1],\
[18.034,46.081,262.155,300.053,300.762,68.747,-1,1,1],\
[18.011,47.06,262.157,299.784,299.931,69.334,-1,1,1],\
[17.941,48.121,262.168,299.433,299.063,69.891,0.27,1,1],\
[19.123,30.468,268.6,312.245,318.328,50.983,-1,1,1],\
[0.573,29.688,274.923,27.015,42.309,322.172,-1,1,1],\

[1.782,46.682,261.715,50.877,61.894,283.797,-1,1,1],\
[3.019,48.307,262.314,52.451,62.72,286.19,-1,1,1],\
[3.287,50.45,267.023,51.845,60.102,287.885,-1,1,1],\
[4.691,53.222,273.25,52.037,56.872,289.596,-1,1,1],\
[5.48,52.466,273.565,52.482,55.637,292.305,-1,1,1],\
[5.849,52.607,275.002,52.224,54.222,293.2,-1,1,1],\
[7.196,53.908,277.377,53.542,53.301,292.63,0.78,1,1],\
[6.338,53.759,277.911,52.042,52.856,294.051,-1,1,1],\
[6.271,51.906,277.99,51.115,51.437,295.672,-1,1,1],\
[6.214,50.84,278.103,50.462,50.558,296.61,-1,1,1],\
[6.078,49.354,278.403,49.364,49.166,298.264,-1,1,1],\
[4.99,41.105,283.964,37.927,40.259,313.325,-1,1,1],\

[348.432,54.379,277.506,348.85,97.47,307.65,0.78,1,1],\
[357.114,52.698,273.62,355.166,91.4,306.2,-1,1,1],\
[357.348,59.221,287.163,355.573,91.807,313.3,-1,1,1],\
[357.327,58.074,284.815,355.53,91.76,312.05,-1,1,1],\
[357.558,55.624,279.347,358.183,89.229,309.066,-1,1,1],\
[357.89,53.724,275.106,0.644,87.287,306.656,-1,1,1],\
[358.854,51.765,268.477,6.189,85.154,301.831,-1,1,1],\
[0.425,48.711,261.528,12.717,82.22,297.269,-1,1,1],\
[2.262,46.285,256.078,19.257,80.045,293.347,-1,1,1],\
[5.09,43.881,250.626,27.77,77.956,288.85,-1,1,1],\
[10.161,41.225,244.793,40.93,75.876,283.192,-1,1,1],\
[15.274,39.584,241.511,52.84,74.468,278.866,-1,1,1],\
[13.839,38.73,242.65,50.933,73.124,280.718,-1,1,1],\
[20.905,37.167,240.398,66.715,71.1,275.168,-1,1,1],\
[25.157,36.621,240.184,76.01,69.93,271.935,-1,1,1],\
[31.037,35.729,241.188,89.155,67.278,267.093,-1,1,1],\
[29.271,36.472,242.598,87.21,66.643,267.959,-1,1,1],\
[36.967,36.621,246.458,106.413,64.441,259.122,-1,1,1],\
[36.944,35.899,246.483,106.483,63.7,258.903,-1,1,1],\
[40.964,36.522,250.362,118.24,63.021,252.313,-1,1,1],\

```
[44.273,36.668,255.476,130.7,61.351,243.88,-1,1,1],\
[43.112,37.527,257.331,129.846,60.116,243.826,-1,1,1],\
[45.54,39.47,261.537,133.206,59.547,243.181,-1,1,1],\
[48.607,41.283,266.599,137.711,58.787,238.707,-1,1,1],\
[48.063,40.215,265.48,137.141,58.411,239.05,-1,1,1],\
[53.897,35.674,255.984,140.784,64.694,240.373,-1,1,1],\
[54.092,39.809,255.761,139.476,67.597,243.911,-1,1,1],\
[53.051,25.52,258.697,145.93,56.477,228.898,-1,1,1],\
[48.351,68.869,330.538,43.9,12.107,317.239,-1,1,1],\
[54.395,77.835,344.616,147.31,7.963,211.047,-1,1,1],\
[54.395,79.887,346.114,144.675,8.246,213.714,-1,1,1],\
[55.9,83.208,347.582,141.465,10.924,217.091,0,1,1],\
[57.369,72.806,323.982,117.364,21.902,242.473,-1,1,2],\
[60.184,49.348,286.833,112.337,35.499,249.514,-1,2,2],\
[3.149,348.471,224.924,92.551,36.172,267.888,-1,2,2],\

]
    for a in range(len(posList)):
        while(stage.value!=posList[a][7]):
            lockNext.acquire()
            print(stage.value)
            print(posList[a][7])
            if(next_stage_1.value==next_stage_2.value & next_stage_1.value!=stage.value):
                stage.value = next_stage_1.value;
            lockNext.release()
            time.sleep(0.2)


        success &= angular_waypoints(base, posList[a], router)
        lockNext.acquire()
        next_stage_1.value = posList[a][8]
        lockNext.release()
```

```
        return 0 if success else 1

def arm2(stage,next_stage_1, next_stage_2, lockNext):
    import utilities

    args = utilities.parseConnectionArguments(ip = "192.168.2.10")

    with utilities.DeviceConnection.createTcpConnection(args) as router:

        # Create required services
        base = BaseClient(router)
        base_cyclic = BaseCyclicClient(router)

        # Example core
        success = True

        success &= example_move_to_home_position(base) # go to home position
```

#input position list for arm 2 below. 7th array index is used to control gripper. 1.0 is fully closed, 0.0 is fully open. if left as -1, will not change gripper. 8th and 9th array indexes are for

#synchronization between the steps. 8th index is step required by both arms before execution, and 9th index is step to proceed to for that arm after executing that position.

```
        posList = [
[98.277,28.822,319.456,88.489,251.482,11.715,0,1,1],\
[100.222,49.636,266.163,87.234,325.225,13.89,0.8,1,1],\
[97.521,49.139,265.575,87.456,325.686,13.625,-1,1,1],\
[97.522,26.357,311.787,88.567,256.666,11.106,-1,1,2],\
[56.506,28.136,317.414,88.524,252.9,282.31,-1,2,2],\
[56.509,25.142,300.836,88.573,266.388,282.7,0.5,2,2],\
[56.509,25.142,300.836,88.573,266.388,282.7,1,2,2],\
```

[55.533,25.051,284.741,88.796,282.4,274.046,-1,2,2],\
[56.235,25.184,302.313,88.586,265.012,282.323,-1,2,2],\
[47.593,24.841,299.78,88.964,267.631,273.822,-1,2,2],\
[49.667,24.114,293.426,88.8,272.953,275.976,-1,2,2],\
[54.442,25.352,295.227,88.652,272.298,280.775,-1,2,2],\
[54.45,25.66,289.247,88.652,278.584,280.876,-1,2,2],\
[54.446,25.867,303.134,88.65,264.913,280.58,-1,2,2],\
[63.722,25.5,302.494,88.319,265,289.754,-1,2,2],\
[63.725,25.015,294.445,88.324,272.474,289.99,-1,2,2],\
[58.29,23.552,292.385,88.523,272.288,284.6,-1,2,2],\
[58.291,24,286.232,88.46,279.866,284.758,-1,2,2],\
[58.291,26.064,312.795,88.456,255.347,284.096,-1,2,2],\
[56.411,15.864,301.927,89.895,256.022,9.556,-1,2,2],\
[56.741,14.297,286.07,89.894,270.346,8.669,-1,2,2],\
[55.568,25.048,300.865,89.934,266.221,7.445,-1,2,2],\
[55.57,24.75,294.222,89.915,272.618,7.456,-1,2,2],\
[54.715,30.181,303.784,90,268.53,6.7,0.78,2,2],\

[54.006,38.793,318.009,91.18,255.2,8.633,-1,2,2],\
[54.008,38.641,316.725,91.177,256.31,8.6,-1,2,2],\
[53.557,46.611,330.366,92.095,244.726,8.6,-1,2,2],\
[53.559,46.23,328.728,92.074,245.99,8.519,-1,2,2],\
[53.315,52.477,339.805,92.744,237.498,8.758,-1,2,2],\
[53.136,58.082,350.139,93.46,229.991,9.2,-1,2,2],\
[52.999,61.726,355.31,94.399,224.679,9.763,-1,2,2],\

[53.136,58.082,350.139,93.46,229.991,9.2,0.746,2,2],\

[53.17,58.796,354.256,93.731,226.483,9.557,-1,2,2],\
[53.232,58.212,354.977,93.786,225.323,9.735,-1,2,2],\
[53.214,58.775,355.494,93.978,224.446,9.794,-1,2,2],\
[53.194,59.297,355.2,93.981,225.093,9.738,-1,2,2],\
[53.186,59.453,354.905,93.839,225.935,9.587,-1,2,2],\
[53.185,59.453,354.904,93.837,229.368,9.587,-1,2,2],\

```
[53.186,59.452,354.906,93.834,232.268,9.587,-1,2,2],\
[53.186,59.452,354.906,93.834,233.952,9.587,-1,2,2],\
[53.338,58.104,358.541,94.191,225.677,10.219,-1,2,2],\
[52.785,33.901,314.576,88.801,258.171,358.99,1,2,2],\
[52.784,32.862,305.722,88.807,266.014,359.162,-1,2,2],\
[52.538,33.522,310.028,88.813,262.429,358.899,-1,2,2],\
[58.806,33.987,310.7,88.949,262,5.2,-1,2,2],\
[58.804,33.596,306.733,88.948,265.663,5.208,-1,2,2],\
[58.425,31.761,323.402,88.815,247.1,100.328,0,2,2],\
[65.84,357.142,232.335,125.591,40.182,65.186,-1,2,2],\
[70.356,21.924,218.76,117.673,74.52,85.971,-1,2,2],\
[70.518,38.583,218.836,116.737,89.32,93.562,-1,2,2],\
[69.254,38.236,223.076,116.473,85.258,87.69,0.7356,2,2],\
[69.102,24.882,222.855,117.465,73.424,81.622,-1,2,2],\
[323.87,44.694,253.488,332.665,104.848,114.5,-1,2,2],\
[323.325,90.655,277.734,328.567,90.816,95.301,0.415,2,2],\
[323.9,38.452,254.396,332.716,105.072,120.8,-1,2,2],\


]


    for a in range(len(posList)):

        while(stage.value!=posList[a][7]):
            lockNext.acquire()
            if(next_stage_1.value==next_stage_2.value & next_stage_1.value!=stage.value):
                stage.value = next_stage_1.value;
            lockNext.release()
            time.sleep(0.2)

        success &= angular_waypoints(base, posList[a], router)
```

```
            lockNext.acquire()
            next_stage_2.value = posList[a][8]
            lockNext.release()

        return 0 if success else 1
def main():

    # Import the utilities helper module
    sys.path.insert(0, os.path.join(os.path.dirname(__file__), ".."))
    stage = Value('i', 1)
    next_stage_1 = Value('i', 2)
    next_stage_2 = Value('i', 2)
    lockCurrent = Lock()
    lockNext = Lock()
    a1 = Process(target=arm1, args = (stage,next_stage_1, next_stage_2, lockNext))
    a2 = Process(target=arm2, args = (stage,next_stage_1, next_stage_2, lockNext))
    a1.start()
    a2.start()
    a1.join()
    a2.join()

if __name__ == "__main__":
    exit(main())
```