Demeter
Bell Pepper Harvester

Capstone Project Report

Team Members:

Amrit Ramesh

Juan Carlos Del Pino

Juan Pablo Bernal

Kevin Lewis

Santiago Delgado

Ziyi Yang

Advisor:

Taskin Padir

Department of Electrical and Computer Engineering

Northeastern University

April 20, 2021

I. Abstract

Red bell peppers are ripe for a short window and during the time while they get to final ripeness they are particularly prone to disease or rot. In order to counteract this, a pepper harvester could increase the speed and accuracy of which peppers are harvested. We designed and created a robot capable of identifying and harvesting ripe bell peppers in such a way that the plant itself is not damaged. To accomplish this, we incorporated a number of design features including a robotic arm, a machine learning model, and computer vision algorithms, and various 3D printed parts. Our requirements are that the system can find, detect and cut ripe red bell peppers with discretion and the ability to not cut other unripe peppers or red objects that are not peppers.

II. Introduction

Team Demeter has designed a computer vision system surrounding the Kinova Jaco arm to autonomously pick ripe red bell peppers. The basis of this project was an understanding of the deficiencies of current farming practices when it comes to bell peppers. More specifically, mechanical harvesters, while very fast, are not very effective at distinguishing between ripe and unripe peppers. Furthermore, red peppers do not ripen all at once so a harvester must be able to tell between ripe and unripe peppers. At the same time, more modern robotic harvesters, while very effective at distinguishing ripeness and therefore generally more precise in picking peppers, have some limitations when it comes to their high cost and their speed. With that said there currently are not any commercially viable robotic bell pepper pickers. Ultimately, while we don't aim to improve on the SOA systems algorithms, hardware, communication, and general speed we do think we can create a rivaled system that is lighter weight, cheaper, and potentially commercially viable.

For the actual design of the pepper picker, we decided to use a net and blade apparatus attached to the Kinova robotic arm such that we could grab, cut the stem, and transport the pepper to a basket. In order to actually recognize the pepper, we used a stereo vision camera module that would not only help us detect ripeness but also help us determine the distance so we can actually pick the pepper. More specifically, as the arm moves, we use one of the cameras to scan through the field of peppers. Whenever the camera detects red, we then stop the movement and use computer vision algorithms to move the arm to get that red object fully within the frame. Finally, we create a bounding box around that red object and send that bounding box image to an ML model that predicts the probability that it is a ripe red bell pepper. In this case, we use the pre-trained ImageNet model trained on the VGG16 architecture. If the red object is a pepper, we will then use the depth of the stereo vision to find the correct 3D position of the target with respect to the arm. The recognition system was run in a Raspberry Pi 4 Model B that communicated using ROS services to a computer running the arm driver. Moreover, a third computer was used to run the machine learning model since the Raspberry Pi was not powerful enough.

To test and validate the system we hung three peppers so that two red would appear in the same frame and a third green pepper would exist to show how the system would handle seeing an unripe pepper. This proved the recognition system can handle seeing two peppers and picking one first, and then the other. This also proved that the stereo vision gives accurate depth, and the hardware can handle cutting through a pepper stem. Results showed that Demeter was able to do everything successfully except cut the stem because the fingers of the JACO arm did not have enough strength.

III. Design

III.i Software

Our software design is composed of two main systems, the recognition system which is in charge of finding and recognizing bell peppers, and the arm driver which is in charge of controlling a Kinova JACO gen 1 arm. First, a master core is initialized and both systems were launched at the same time. The arm driver started in scanning mode, this took the arm around the plants in a lawnmower pattern until a pepper candidate was found. While the arm was in this mode a camera at the end effector was recording video, processing each frame looking for a red mass. Once a red object was detected, the recognition system forced the arm out of the scanning mode and took a final image for processing. Then, If there are multiple objects in the frame, the algorithm would target one and flag the others. If this object was cut off, the arm would move until it had the entire pepper candidate inside the frame. Afterward, this object was sent to an ML model for validation. If it had a probability greater than 80% of being ripe red bell pepper, using stereo vision the 3D coordinates of the object were calculated. Finally, these coordinates were sent to the arm to harvest and transport the pepper to a secure location. If there were flagged candidates the arm would go back to the location where they were found and go through the recognition process all over again. Moreover, if there weren't any more candidates, the arm would go back to scanning mode to look for more candidates. This would continue until the arm scanning pattern was complete.

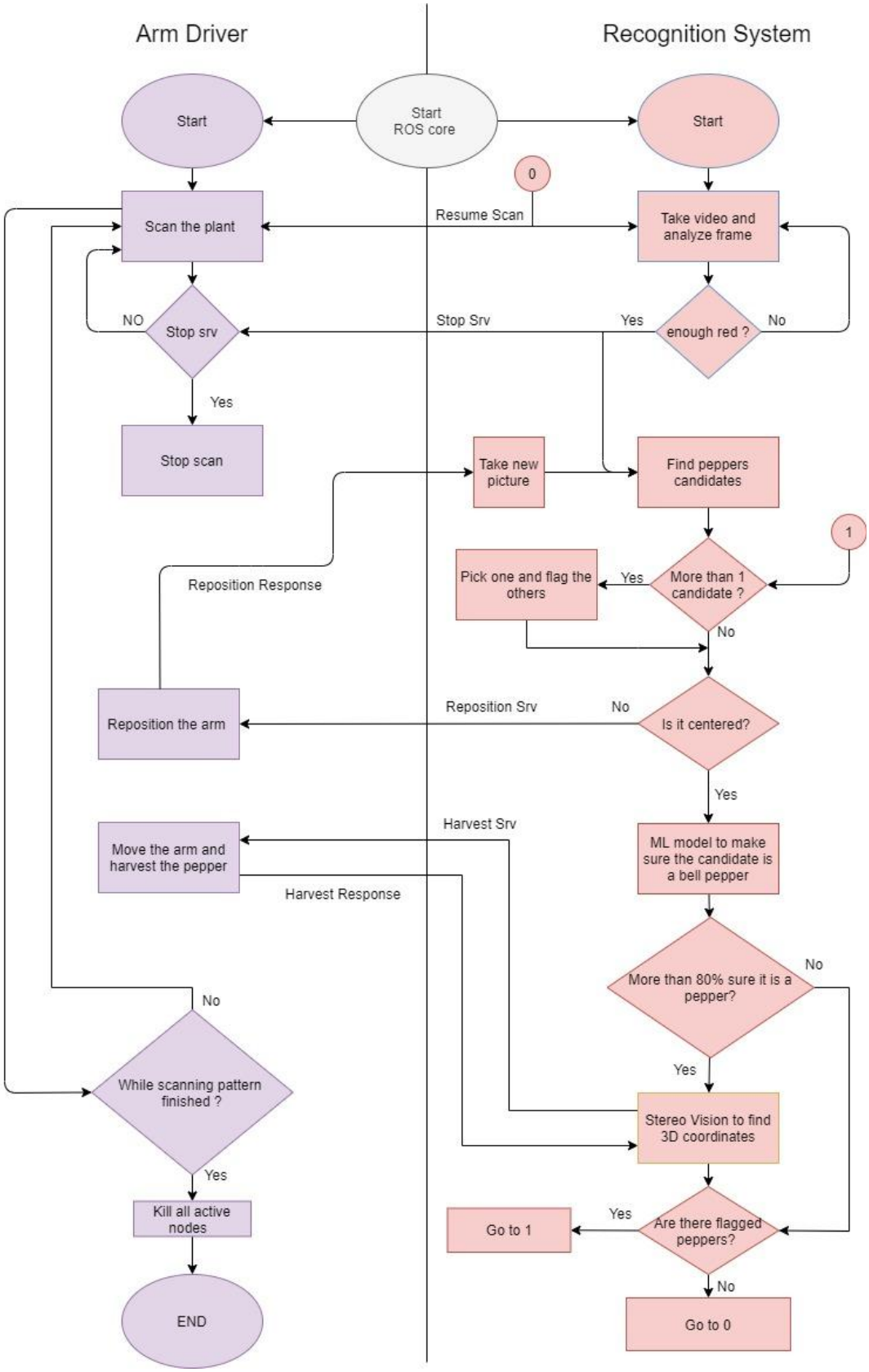


Figure.III.i.1 General Approach Flowchart

III.i.1 ROS Communication

Both systems used ROS to communicate with each other. Due to preexisting libraries and software constraints, two different versions of ROS were used. The Kinova JACO arm libraries were designed to be run with ROS Melodic. However, the recognition system used TensorFlow for the ML model and Python3 for better image processing libraries. Given these constraints, ROS Noetic was used to run the recognition algorithms. Each system was run on a different device, the arm driver was running on a Mac laptop with a Linux environment and the recognition system was running on a Raspberry Pi 4 Model B. Moreover, since the Raspberry Pi didn't have enough power to run the ML model a third Windows 10 laptop was used. To communicate between the three different devices, all were connected to the same network that was hosted by the Windows laptop as a mobile hotspot. Each machine had its own IP address inside the local network and it was used to identify ROS nodes, servers, and clients. This was achieved by setting the environmental variable ROS_IP to the value of each machine's IP address. Moreover, a single ROS core was running locally in the Mac laptop and all other devices connected to it by setting the environmental variable ROS_MASTER_URI to http://<Mac_IP_Address>:11311. This ensured that each ROS script was able to find a ROS core and that all nodes were connected to the same core.

There were three main nodes running, one in each machine; the recognition system node, the arm driver node, and the ML model node. These nodes used ROS services to send and receive important information about what to do next. To achieve proper functioning of the three nodes five services were used:

1. Start: The server was the recognition system node and the client was the arm driver node. This service helped to synchronize the 2 nodes. The arm driver would take the arm to the initial position and wait for this service to be online before beginning to scan. The recognition system would only initialize this service once everything was set up and the camera was ready to start recording. This service was also used to kill the recognition node once the arm was done scanning.

2. Stop: The server was the arm driver node and the client was the recognition system node. This service was used to signal the arm driver when a pepper candidate was found. The arm would exit scanning mode and return its current position to the recognition system.

3. Reposition: The server was the arm driver node and the client was the recognition system node. This service was used when a reposition of the arm was needed. This happened when the arm was trying to center the candidate on the frame or when it was trying to return to a previous location. Here the recognition system would send a set of new coordinates and the arm would only respond with a boolean acknowledgment.

4. Harvest: The server was the arm driver node and the client was the recognition system node. This service was very similar to reposition srv. However, this time the arm would enter harvesting mode which would go to the coordinates sent by the recognition system, cut the stem, and finally transport the pepper to the basket location. The response was a boolean acknowledgment to signal when the harvesting was done.

5. ML_Model: The server was the ML model node and the client was the recognition system node. This service was used to validate that a pepper candidate was in fact a ripe red bell pepper. Once the candidate was fully in the frame and a bounding box was generated around it, it was sent to the ML model. The response was the percentage returned by the model of how much it believed the input image was a bell pepper.

III.i.2 Stereo Vision

For the recognition system, we used a 960p duo lens stereo USB camera module shown in Figure III.i.2.1. The two cameras are synchronized by the manufacturer which would give us a photo that contains both views from the left and right camera. Our main goal is to get the depth map from the stereo vision. So before all, we had to calibrate the two cameras.

Figure III.i.2.1: USB Stereo camera module

A depth map requires the use of two cameras with their optical, vertical, and horizontal axis all in parallel. However, in the real world, cameras are different and impossible to align perfectly. We used an open-source python library called stereovision which could help calibrate the two cameras and find matrices for correction. Firstly, we took 30 pictures of someone holding a printed chessboard shown in Figure III.i.2.2. Then we used functions from the stereo vision library to detect the chessboard and calibrate the two cameras.

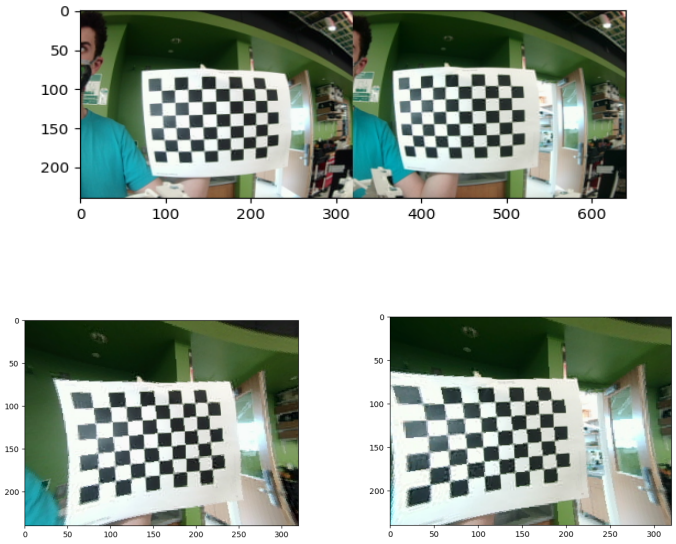
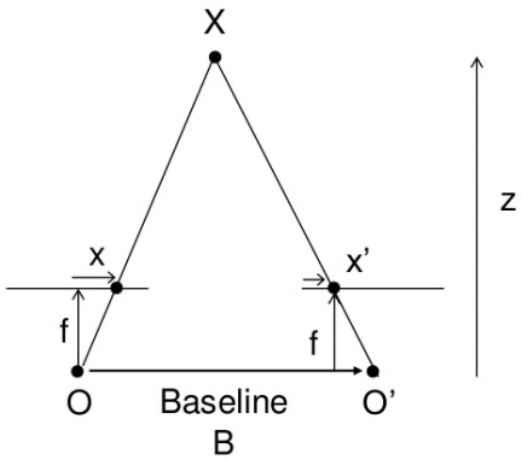


Figure III.i.2.2: Chessboard photo example and calibration results

Stereo vision techniques use two cameras to see the same object. The two cameras are separated by a baseline distance. The two cameras simultaneously capture two images. The two images are analyzed to find the differences between them. Essentially, one needs to accurately identify the same pixel in both images, known as the problem of correspondence between the two cameras. Features like corners can be easily found in one image, and the same features can be searched for in the other image. Then we then had to find the disparity map. The disparity is the difference in image location of the same 3D point when projected under perspective to different cameras. The disparity helped to get the depth information of the scene and ultimately reconstruct the points in the 3D world. If we have two images of the same scene, we can get depth information using triangulation techniques. Figure III.i.2.3 is an image that shows how stereo vision works, and the mathematical formulas used to determine 3D coordinates. x and x′ are the distance between points in the image plane corresponding to the scene point 3D and their camera center. B is the distance between two cameras (which we measured is 60mm) and f is the focal length of the camera (3mm). Then we could calculate the depth by using this equation *depth = (baseline * focal length) / disparity.* So in short, the equation says that the depth of a point in a scene is inversely proportional to the difference in distance of corresponding image points and their camera centers. With this information, we can derive the depth of all pixels in an image.



$$disparity = x - x' = \frac{Bf}{Z}$$

Figure III.i.2.3: Picture showing how stereo vision works

Then the code could generate a real-time depth map by using the OpenCV functions. And the code also gave us a modifier that we could adjust the sensitivity of the depth view as shown in Figure III.i.2.4. There are some parameters, and you may need to fine-tune them to get better and smooth results. Parameters:

- texture_threshold: filters out areas that don't have enough texture for reliable matching
- Speckle range and size: Block-based matches often produce "speckles" near the boundaries of objects, where the matching window catches the foreground on one side and the background on the other. In this scene, it appears that the matcher is also finding small spurious matches in the projected texture on the table. To get rid of these artifacts we post-process the disparity image with a speckle filter controlled by the speckle_size and speckle_range parameters. speckle_size is the number of pixels below which a disparity blob is dismissed as "speckle." speckle_range controls how close in value disparities must be to be considered part of the same blob.
- Number of disparities: How many pixels to slide the window over. The larger it is, the larger the range of visible depths, but more computation is required.
- min_disparity: the offset from the x-position of the left pixel at which to begin searching.
- uniqueness_ratio: Another post-filtering step. If the best matching disparity is not sufficiently better than every other disparity in the search range, the pixel is filtered out. You can try tweaking this if texture_threshold and the speckle filtering are still letting through spurious matches.
- prefilter_size and prefilter_cap: The pre-filtering phase, which normalizes image brightness and enhances texture in preparation for block matching. Normally you should not need to adjust these.

After tunning, the final depth map worked perfectly as when the detecting object is close to the camera it would be shown in radish color and when it's further away the color would be lighter to yellowish as shown in Figure III.i.2.5.

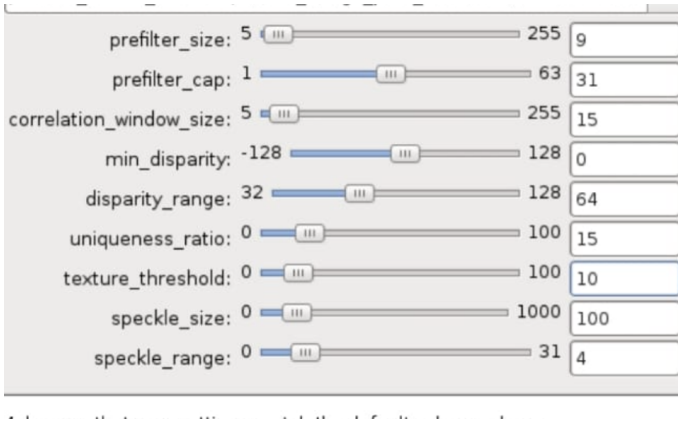


Figure III.i.2.4: Depth map modifier

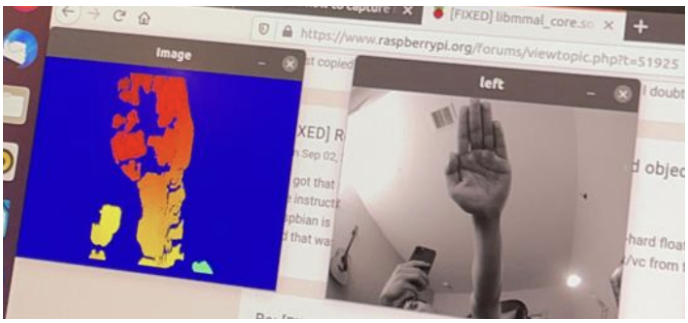


Figure III.i.1.5: Depth map result

With the depth information, the 3D X and Y coordinates could be found using the equations $X_{3D} = \frac{(x_{img} - c_x) * Z_{3D}}{f}$ and $Y_{3D} = \frac{(y_{img} - c_x) * Z_{3D}}{f}$ where X_{3D}, Y_{3D}, and Z_{3D} are the 3D coordinates of a point, f is the focal length of the cameras, c_x *and* c_y are the pixel coordinates of the center of the image and x_{img} and y_{img} are the pixel coordinates of the point on the left image. Finally, with all three coordinates, the location of a single point with respect to the camera was found. The last step was to find the homogeneous transformation matrix that would compute the coordinates of the same point with respect to the end-effector of the arm. The position of the camera with respect to the end-effector frame was measured. With the current setup the camera was located at x = -0.01m y = 0.12m and z = -0.05m. Moreover, the axes were not aligned. The camera's x-axis corresponded to the arm's y-axis, the camera's y-axis corresponded to the arm's z-axis, and the camera's z-axis corresponded to the arm's x-axis.

III.i.3 Recognition System

As stated in the intro we initially devised a plan where the basis of the recognition system worked in 3 main parts. First, the arm, with the camera attached, would scan the field of peppers in a lawnmower pattern. When the recognition system detects red it sends a signal to the arm to stop. At this point in time, two possibilities could exist: either the pepper candidate is fully in frame, or the pepper candidate is not fully in frame. There would need to be an algorithm to guarantee the pepper candidate was fully in frame, or move the arm so that the pepper candidate would fully be inframe. The final step would be to check if the candidate is truly a ripe red bell pepper At each step of the process we devised the algorithms and tools needed to make it work smoothly and cohesively.

The first step of moving the arm with the attached camera in a lawnmower pattern was simply done through the arm controller. During that process of moving up and down the camera captures a video. As it captures each frame, we used CV2's suite of python tools to check whether red reaches a certain threshold (based on the pixel values in a certain range) within a frame of the video and based on the output, if there was enough red, send a stop command to the arm. At this point in time, the camera changes modes from taking continuous video to taking single capture images to make the next step of localizing the candidates easier. At the same time, in order to make sure the arm didn't move after the red was detected we also incorporated a feedback loop in the recognition system that would send reposition coordinates to the arm to get back to where sufficient red was found within a frame of the video.

Figure III.i.3.1 Lawnmower Pattern

For the next step, of checking whether the red candidate is within the frame, we had a couple of options of how to proceed. We thought of using existing deep learning tools that would be able to draw bounding boxes around the red candidate. We could then, based on the bounding box dimensions, determine if either the red candidate was cut off, or if it was even a pepper. If it is determined to be cut off based on the dimensions (and its closeness to any edge of the image frame) then we can move the arm appropriately. But, even though the deep learning systems that would create the bounding box around the candidate have been proven to be very accurate, using just the bounding box dimensions to determine whether the candidate is actually a pepper or cut off didn't seem to be the most accurate thing to do. Furthermore, running an image through a deep learning algorithm at each frame, as we try

to localize the candidate, would likely be very computationally expensive to run on the Raspberry Pi. Therefore we chose to use a methodology rooted solely in computer vision.

More specifically, at the starting frame, where we initially stopped the arm, there may be one or more distinct red candidates; at least one since we stopped movement after detecting red, and more than one if multiple candidates are near each other. At this point we thresholded the image for red to help ignore the interfering objects in the background as well as any unripe or half-ripe green peppers in the vicinity. We then use CV2's findContours function to create contours around the candidates within the threshold frame. The contour finder function, based on Suzuki and co.'s 1985 paper on Topological Structural Analysis Appendix A.4, works by looking for drastic changes of color between two pixels and marking down one of them as the "contoured edge". This means that if the thresholding didn't work as expected, due to some light exposure that caused a red candidate to be partially thresholded out, the contour for that candidate will look erratic. Therefore in order to correct this, we ignore all contours that are too small (too few consecutive pixels marked down as a "contoured edge") or are inside a bigger contour. And though this has the side effect of dropping potential red candidates, we believe that it is worth it because it helps deal with the larger issue of variance caused by inconsistent lighting. Figure III.i.2 shows an example visualization of this process.

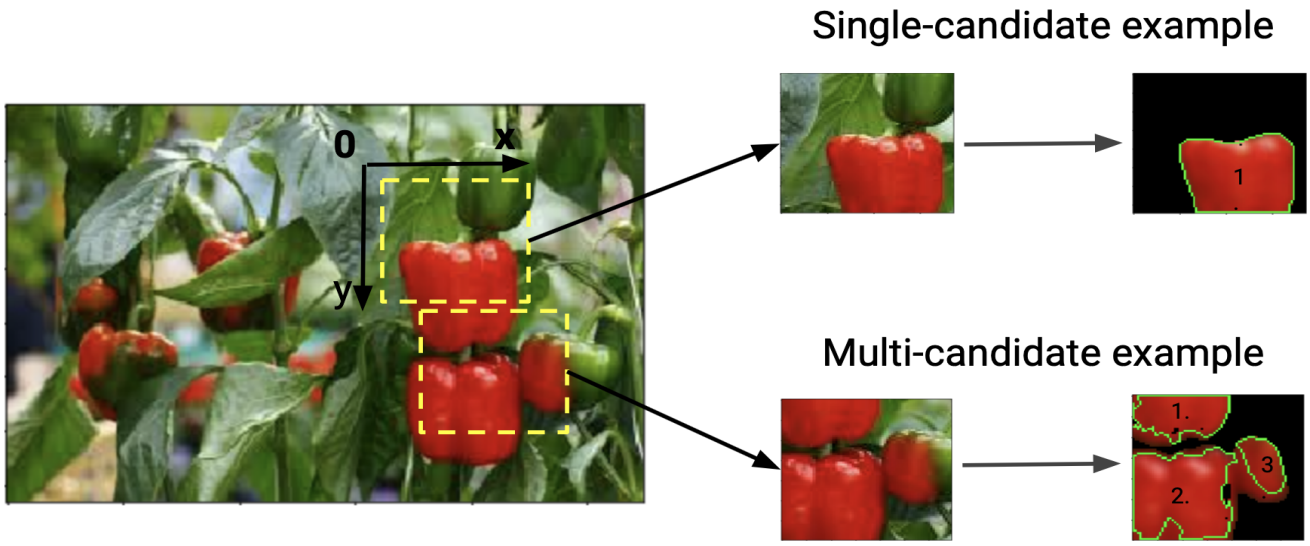


Figure III.i.3.2 Contouring of Initial Stopping Position

As you can see if the initial frame (where the arm initially stops) is the top example frame then the recognition system will threshold the for red and contour the single red candidate. Furthermore, it also ignores the green pepper at the top right completely (which is one mechanism to not pick unripe peppers). At the same time, if the initial frame is the bottom example then we threshold in the same way and are able to draw contours around all three red candidates. Here we also see how issues with lighting can cause the thresholding and therefore the contouring to not be as accurate.

From now on let's use the multi-candidate example to dive into how the localization algorithm works. When we have more than one contoured candidate within the frame we iterate over each of them and check if it is in the frame or not. For example, for the first contour, at the top left, we check if there is a line of contour pixel coordinates at the min/max x or y dimension of the image frame. In this case, there is a long line of contour pixels leveled of a y = 0, and a smaller line of contour pixels at x = 0. This means that the candidate is considered to be cut off at the top but not the left since there not enough contour pixels there to be considered fully cut off. Ideally, at this point, we would like to move the arm (and the camera attached to it) a little up. Ultimately, we chose to reposition the arm 5 cm in cut off direction because after testing it limited overshooting while also not being so small that the movement created an insignificant movement.

At this new position, we then drive the camera to take another picture and go through the same process of thresholding, contouring, and repositioning the arm based on if the target candidate is still cut off. One caveat to this process is that, since we may encounter new red candidates after repositioning and taking another picture, we use the dimensionality of the contour of the target candidate (max height and max width) plus the repositioning from the previous step to keep track of which red candidate in the new frame is the target candidate we should continue navigating towards. After going through this process zero more to many more times, if we reach a position at which the target candidate is deemed to be fully in frame i.e. there is no line of contour pixels at the min/max dimensions of the image frame we stop the cycle. Figure III.i.3.3 is a visualization of the localization process for the first contour.

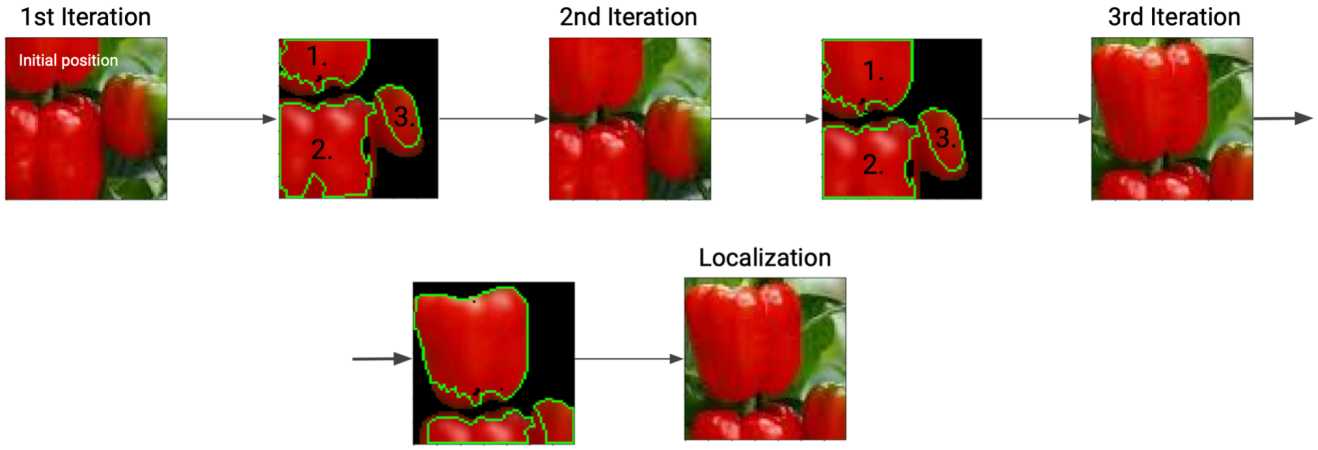


Figure III.i.3.3 Iterations from Initial Position for Localization

As you can see in the first iteration, after we go through the thresholding and contouring process, we reposition only upwards since there are not enough contour pixels at the left side of the image frame for the target candidate to be considered cut off in that direction. At the second iteration, again after we go through the thresholding and contouring process, we see that the candidate is still cut

off at the top, and this time there are enough contour pixels at the left of the image frame for the candidate to be considered cut off in that direction. Because of this, we reposition the arm to move both upwards and left. Finally, at the third iteration, we go through the threshold and contour process once more and see that the target candidate is not cut off in any direction. Therefore we have finally localized target candidate 1.

At this point in time, after localizing the first contoured candidate, we have to check whether it is actually a pepper. To do this, we have employed a pre-trained ML model trained on the ImageNet open-source image data set for the ILSVRC competition. This model is based on a "very deep convolutional neural network" dubbed the VGG16 model architecture. It is trained on ~1.2M images from the ImageNet source library to be able to classify an image into one of ~1000 categories one of which is a bell pepper (regardless of color). See Appendix A.6 for more details. In order to get consistent results from this model we took the fully localized candidate and contoured it once more. Using the minimum and maximum contour pixel coordinates in both the x and y direction we generated a bounding box around the candidate and sent the image within that bounding box to the ML model. This ensured that the objects in the background wouldn't sway the prediction too much. Finally, we resized the bounding box image using CV2 to fit the dimensionality needed for the input of the neural network (224x224). Here is a visualization of this ML process:

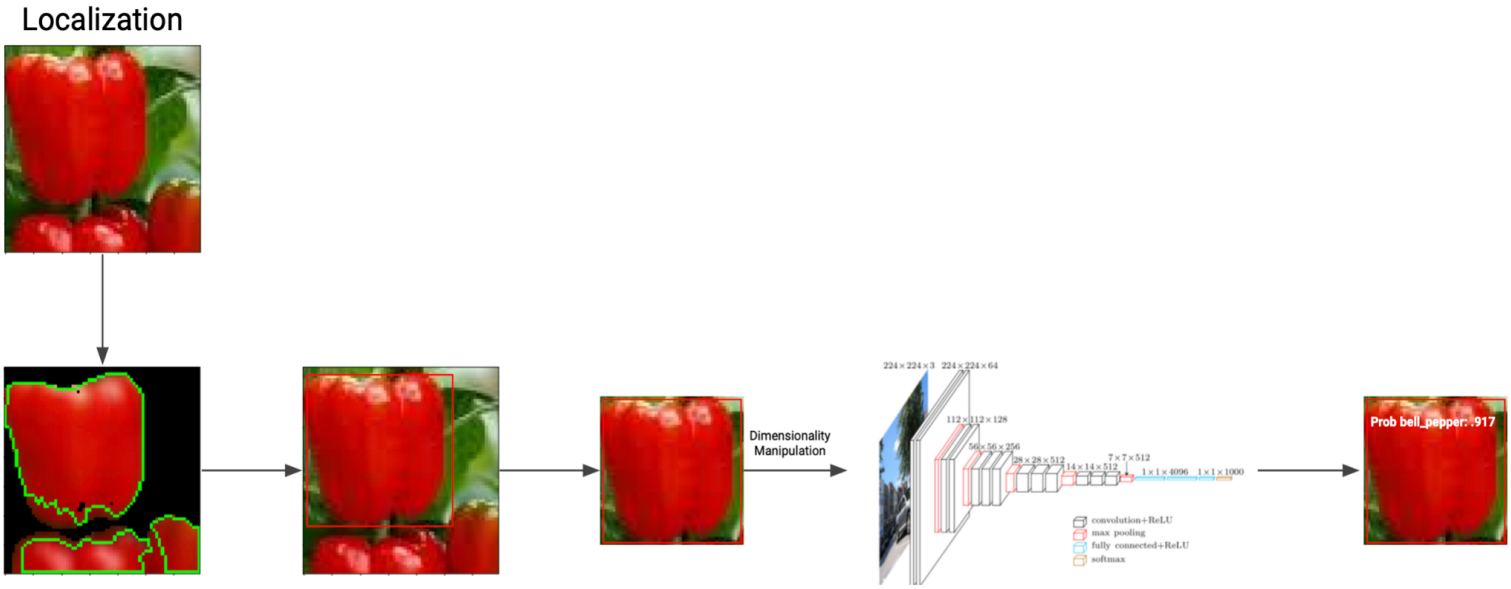


Figure III.i.3.4 Machine Learning Prediction of Candidate

If the predicted probability, that the bounding box image is a pepper, is above 0.8 we then move to the process of harvesting the pepper. The way we do that is by taking the middle top pixel of the bounding box within the full image (we are making the assumption that is where the stem is) and using stereo vision to translate that pixel coordinate into a real-world X-Y-Z coordinate. We can then send

these real-world coordinates to the arm such that it will be in the ideal position to cut the stem of the pepper. At this point, or if the candidate was not considered a pepper earlier, we continue through the iteration of the other contoured candidates in the initial stopping frame by repositioning the arm to where it initially stopped. For example, for candidate 2 we will go through the same process of trying to localize it and send a bounding box image to the neural network to predict if it is a pepper or not. After fulfilling that decision we do the same thing for candidate 3.

The only caveat through this process is, if at the original position the candidate we are navigating towards is too small, we will not go through the process of trying to localize it. In this case candidate 3 was considered too small at the starting position so we ignored it. This is another way we control for not picking half ripe peppers, since if a half ripe pepper doesn't have enough red we won't try to localize it. Here is the remaining process visualized:

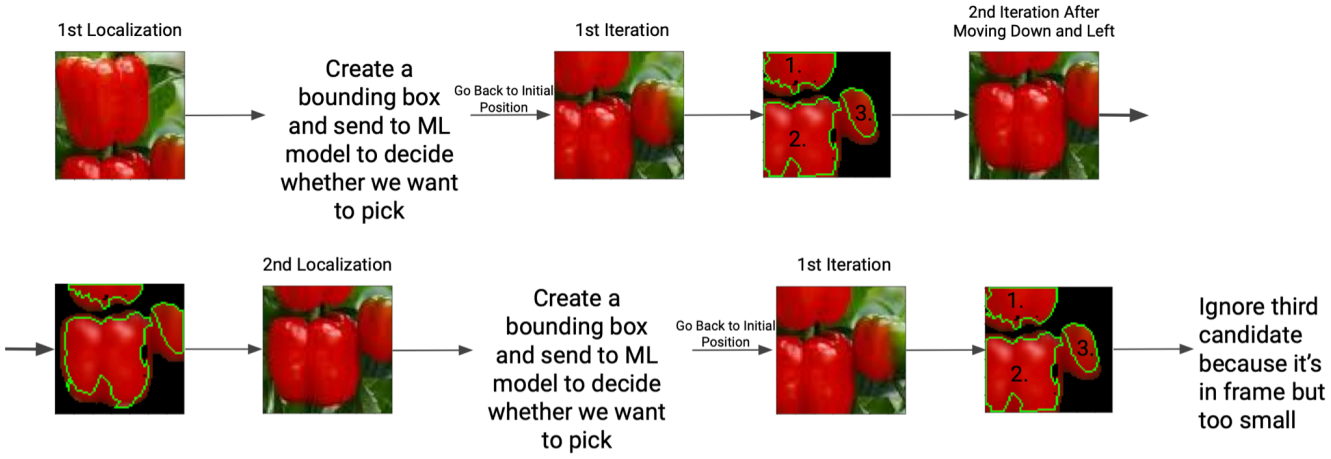


Figure III.i.3.5 The Remaining Process of Localizing other Candidates

After going through this process and picking some or all of the pepper candidates within the initial frame we continue the lawnmower searching pattern and restart the camera video until we see sufficient red again at which point we will repeat the process again. One problem with this methodology, however, is that if we ignore a pepper candidate either because the thresholded red was too small or if the neural network predicted the localized candidate as likely not a pepper we may encounter it again after we continue scanning. In future iterations, we would like to figure out how to save the positions of candidates that should be ignored.

III.i.4 Arm Driver

The arm driver has the role of moving the JACO arm. It has to move to cover all the plants within its reach, move as the recognition system tells it to make peppers candidates fully in frame and harvest peppers at locations designated by the recognition system. The arm driver takes advantage of an

open-source ROS driver made by Kinova Robotics cited in Appendix A.5. This repository takes care of managing the motors to move the arm into the location sent from the arm driver.

To decide what location to move the arm to the arm controller checks a series of booleans. The arm controller accepts server requests from the camera driver and these influence the booleans and the goal location that we are storing and trying to move to. There are a few different movements: Stay in the same spot, move by camera, move by scan, move to basket, and move to the last cut location. Stay in the same spot means we do not send anything to the arm driver. Move by camera uses an internal variable that is set by the reposition or harvest services. Move by scan iterates through an array of locations that move the arm in a lawnmower pattern to cover where the arm can reach. Move to basket moves to the location that is designated as the basket, and move to last cut moves to the location where harvest was sent from. The flow chart in Figure III.i.4.1 shows this path and the effects of the different services.

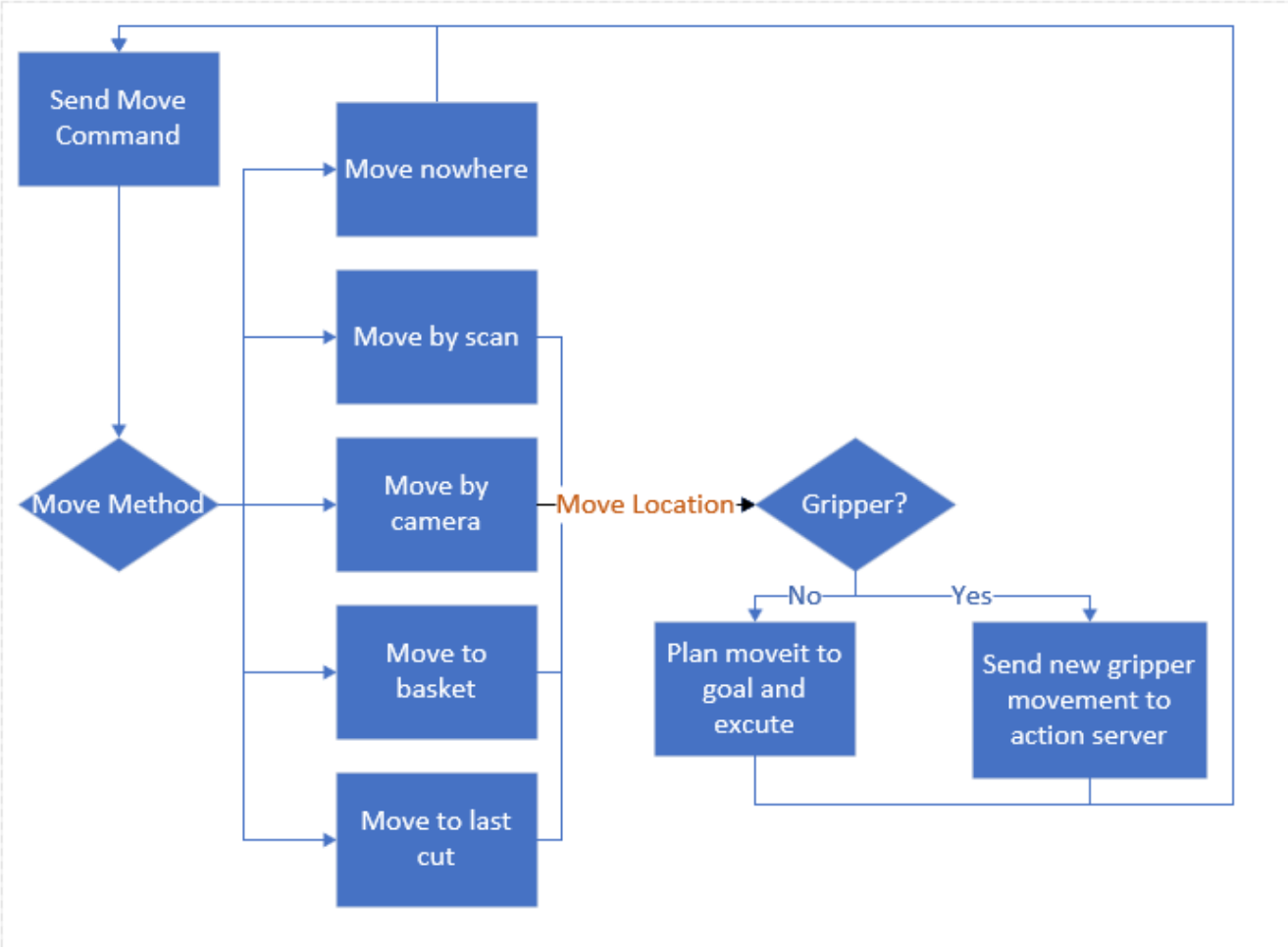


Figure III.i.4.1: Flow Chart of the main drive loop

The initial design used an action server to send commands to the arm. The issue with this was the arm would do straight-line planning, and did not have anything prepared for pathing around collisions with itself. This means if the line would bring the arm in such a way it would hit itself the path would label itself as failed and not be able to avoid the collision area, or move around it successfully. To get around this problem we thought about implementing a random rapid-exploring tree of known locations that have small planned paths that are known to be safe. After some brief investigation into this, it was deemed to be more work than we had time for so we switched to use MoveIt. MoveIt was included in the driver with the URDF files. This made it straightforward to get the arm to move where it is directed to go.

After switching to MoveIt we were left with a few more decisions: how to handle the controller asking to go to places that we cannot reach, what is the acceptable displacement for marking in a certain location, and orientation of the end effector. To handle going to locations that were not feasible we kept the scan a bit inside of the radius of where we can reach to make sure that the reposition and harvest commands would not move the arm outside the range. If we received a command outside of what is reachable this would have caused issues, and would be breaking. The displacement value from current location to a goal location was adjusted over trial and error because if the move command was under a short enough distance the arm would not move, but we needed pretty high accuracy for the cut command so this was hand-tuned to work for both. To handle the orientation of the end effector it was required that the camera stayed level for the recognition system to work properly. To do this the arm was moved into an optimal orientation and that was recorded and became the standard input.

The arm handled service calls by changing internal booleans corresponding with the flowchart of Figure III.i.4.1. These changes match what is shown in Figure III.i.4.2. The main nuance has to do with the harvest service. When it gets called we move by camera, but check if harvest is set and if it is the arm will close the grippers when it gets to the goalLocation. At this point it is set to move to the basket. Once it arrives at the basket it will open the gripper and move to lastCutLocation which is set when the harvest command comes in. When it returns to the last cut location the arm in move by camera mode so that it can properly handle there being more peppers in the frame when the cut happened. If there are not more peppers the reposition will send a stop service with False for the value.

Service	Internal changes	Response
stop	bstop = req.Action {True means don't move} if bstop == False { bbyScan = True bbyCamera = False}	currentLocation and True
harvest	bharvest = True bbyCamera = True bbyScan = False fingerTurn = maxTurn {this makes sure the next time the gripper moves it closes} goalLocation = req.Location lastCutLocation = currentLocation	True
reposition	bbyCamera = True bbyScan = False bstop = False goalLocation = req.Location	True

Figure III.i.4.2:What changes when the arm driver receives services

The final thing the arm driver does is designate when we have finished working. It does so by sending the start service when it has finished going through the scan routine. This is where a future group could implement a mobile base and use this to move forward, or scan the other side if there are two rows of peppers.

III.ii Hardware

Mechanically, Demeter the harvester was designed to fulfill the required tasks in collecting peppers: 1) being able to navigate a red bell pepper field; 2) having a camera, or another method for object detection, with a wide range; 3) the robot has to be able to place the pepper in a container after successful harvesting. The images below, Figures III.ii.1 through III.ii.3, depict the initial stages of the design, improving over the previous one.

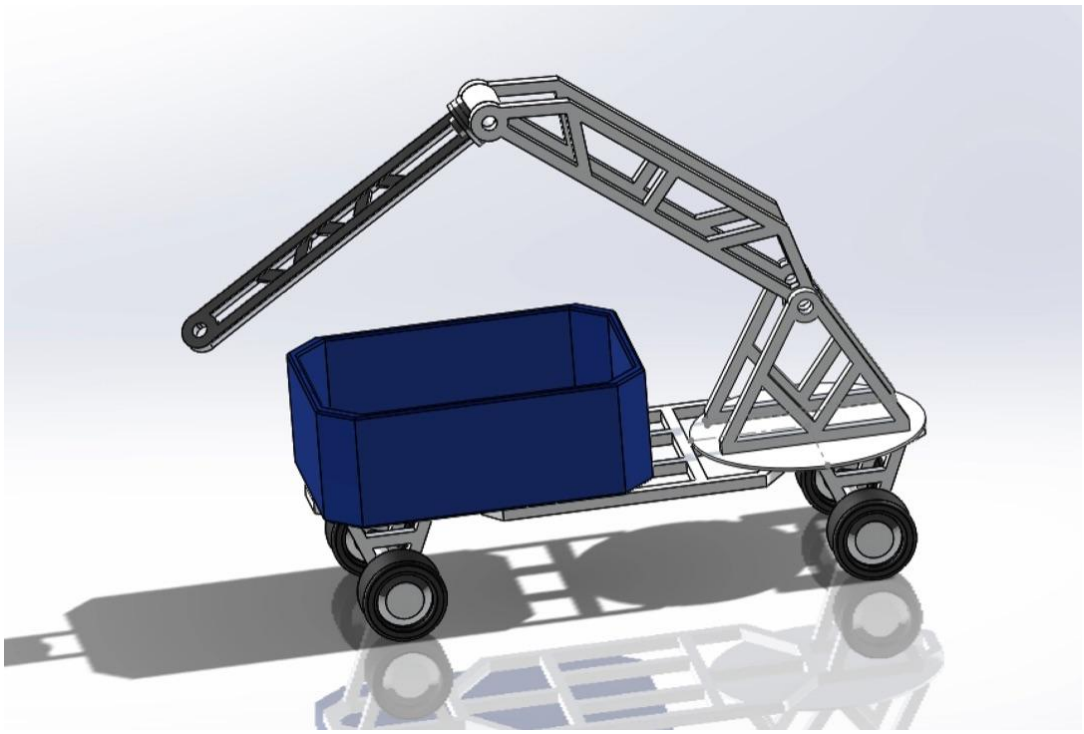


Figure III.ii.1 - Harvester Body Design 1

The body of the harvester as seen in Figure III.ii.1 fulfills the requirements for the harvester, being 12’’ in width to be narrow enough to pass through the plants on the fields. It has the arm coming from the back to be able to reach all the peppers in front, and to the sides of the cart, and easily deposit them in the container. One major flaw this design has is the length of the arm; since the arm is located on the back of the car, it requires a longer arm and therefore stronger motors. It is important to note that the design of the robot is not finished since this was a concept idea.

Figure III.ii.2 - Harvester Body Design 2

The harvester improved in the second design by placing the arm on the front of the car to reduce the torque needed, and the arm also uses a prismatic joint to raise the arm component up and down. While this arm greatly improved on the torque and overall strength of the harvester for the tasks, the major trade-off became the time it would take for the harvester to drop the pepper in a container behind once the pepper is collected. This design seemed like the correct solution, but once the team started putting together the numbers for the budget required, we realized that it would be too expensive to build the arm and cart. It is important to note as well that this harvester design is not finished since it was a concept idea to show the different approach.

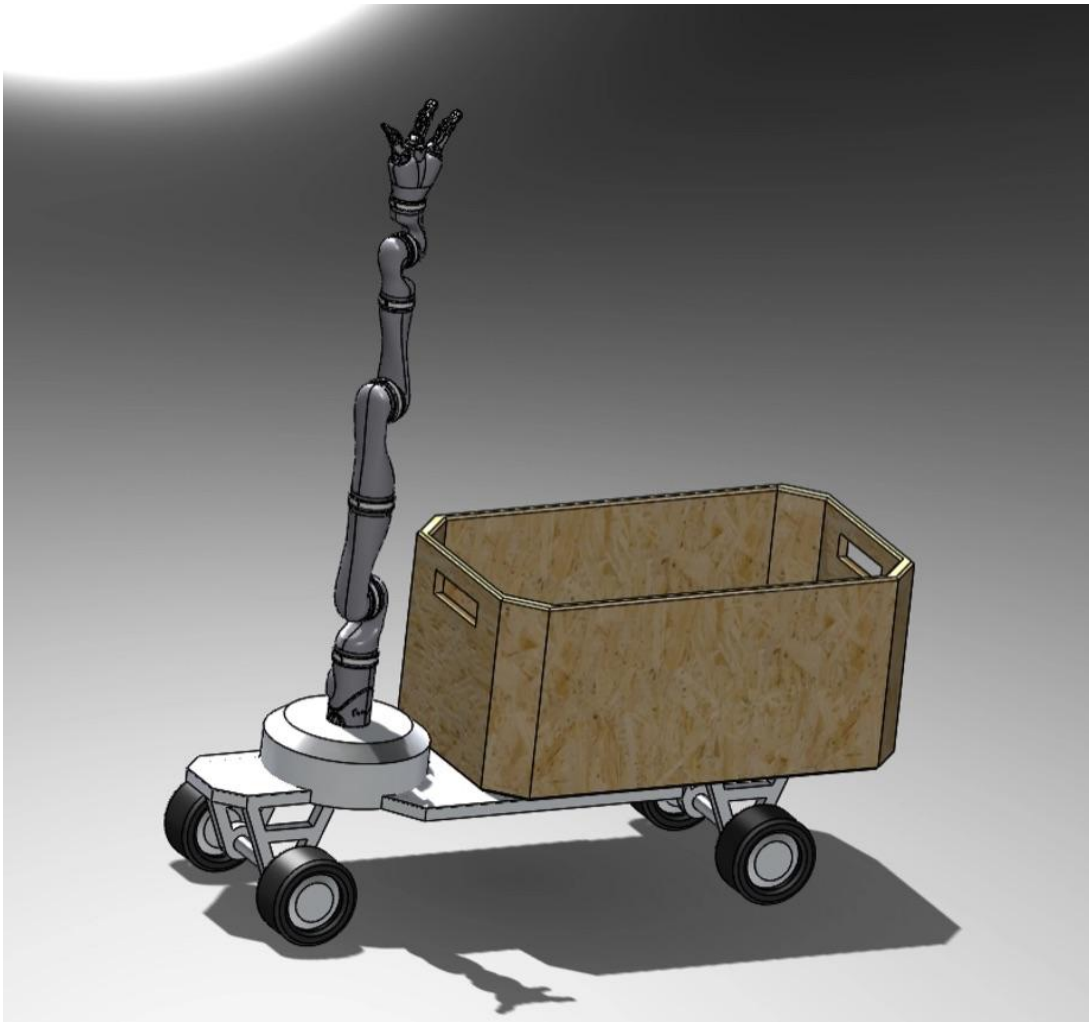
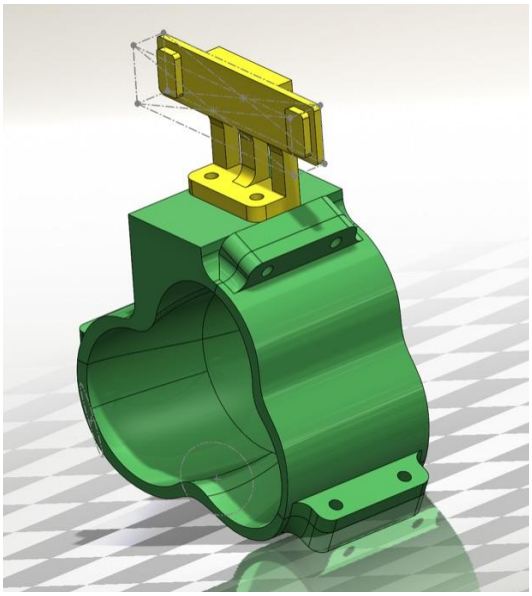


Figure III.ii.3 - Harvester Body Design 3.

The third approach the team took with the harvester was to incorporate an arm we could borrow from Northeastern University and incorporate it into the car as shown in Figure III.ii.3. This constrained the freedom of designing the robot to our needs, and it now required the creativity of creating new parts to make old components do the harvesting job. The advantage to this solution, which ultimately made it the correct approach, is that this fit better within our budget, but it was not enough. Because of this, the team concluded that the harvesting component of the robot was the most important, and took the decision to stop further developing the cart.



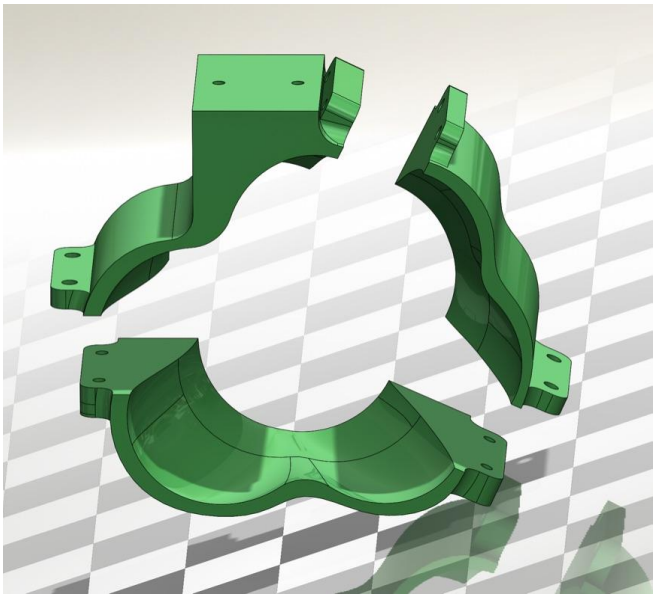


Figure III.ii.4 & III.ii.5 - Kinova Negative for Camera Attachment (Normal and Exploded).

The first approach to developing the fixtures for the arm was to create solid bodies that had the negative surfaces of the robot to be able to strap components very accurately. Figures III.ii.4 & III.ii.5 above shows the second version of the negative to hold the camera at the end-effector. This mechanical design made the camera parallel with the plane of cutting where the fingers travel. The advantage of this design was to accurately know the exact position of the camera and be more accurate with the depth readings. There was a halt put into developing more negatives after the team printed the first set. The arm that was given to us is generation 1, which has been discontinued by Kinova and therefore the team could not access precise CADs of the arm, ultimately defeating the purpose of negatives.

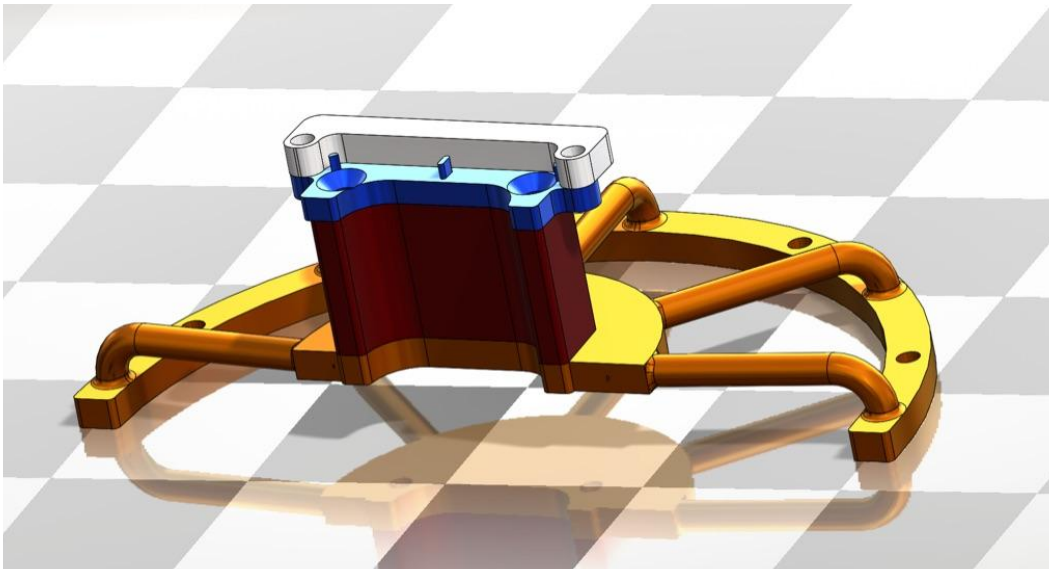


Figure III.ii.6 - Static Harvesting Fixture Top.

After it was found out that the team would not have access to the CADs of the Kinova robot, the focus became to create something easily attachable to the robot that would guarantee the cutting and collecting of the pepper. The fixture breaks into three main components, red, blue/white, and orange pieces. The red component is the part that attaches to the robot. The back face of the red part attaches to each of the fingerprints of both fingers that are used for cutting. Like this, when the fingers close, the red parts come face-to-face creating an empty section in-between the red pieces where the stem of the pepper will go through. The blue and white parts above are where the razor blade goes in; the white component is used to put pressure on the razor blade to keep it in place. The orange part below is where the net attaches to. By having the blade and net in this order, the pepper falls into the net after being cut above. Figure III.ii.7 below shows a fully-assembled blade and net fixture for one finger:

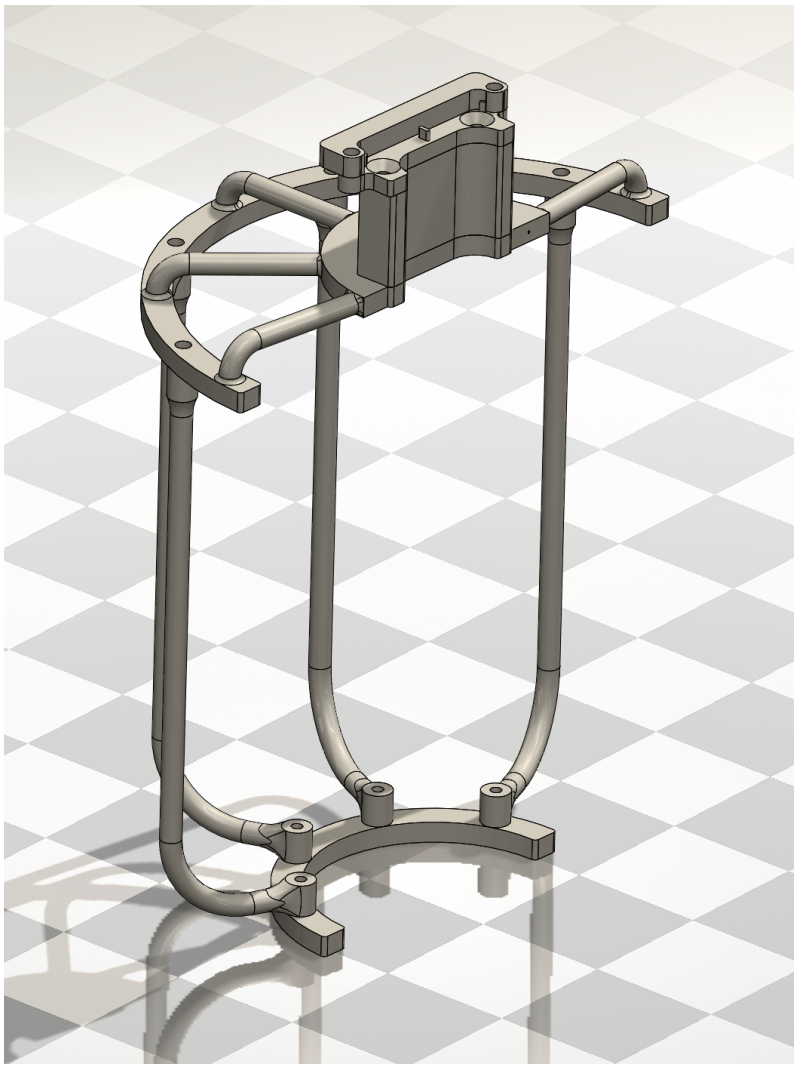


Figure III.ii.7 - Static Harvesting Fixture (Full).

Figure III.ii.8 - Static Harvest Fixture.

The final, 3D printer fixture is seen in Figure III.ii.8 above, and in Figures III.ii.9 & III.ii.10 below. The fixture was lightweight enough that it did not affect the movement of the arm, and the double-sided tape managed to keep it adhered to the robot. Figure III.ii.8 above shows how the blades look once the fingers close. One major issue that was created by this design, is that the blades require to close on the same plane and touch to be able to cut the stem. Due to the weight of the design, the fingers closed to a different, mismatched position each time. Due to this error, the stem of the pepper could not be fully cut. Figures III.ii.9 & III.ii.10 show the harvester and the setup of the “pepper plants”.

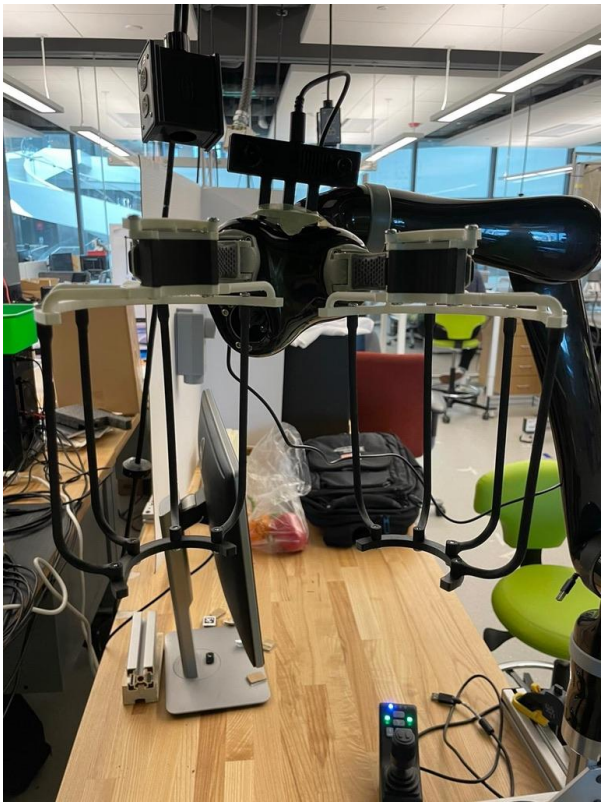


Figure III.ii.9 - Harvesting Fixture (Open).

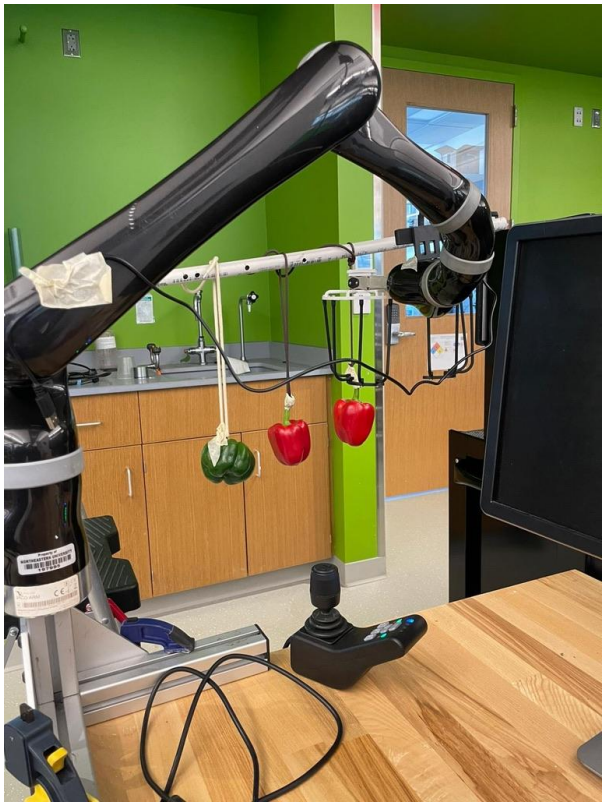


Figure III.ii.10- Scanning Peppers.

IV. Results

In order to test all the parts and how they communicate with each other, we planned on testing with a real pepper plant. Unfortunately, the plants were not ready yet for picking from so we ended up hanging peppers at different heights from a makeshift setup shown above in Figure III.ii.10. We, however, purposely went for peppers with long stems so that we could cut them to replicate what the real world would be like. The arm had the full hardware setup and we had two red peppers that would come into frame at the same time to prove that the contouring methodology works properly. Then later on in the scan it would see a green pepper and ignore it. Pictures from the scan are shown below in Figure IV.1. As the camera is scanning it first stops with the pepper on the bottom of the frame. It centralizes on the left one and picks it then comes back for the second and picks that one. When it tried to pick them the stereo vision was able to line the blades up with the pepper stem accurately. The main issue we had was that the blades were unable to cut through the stem of the pepper. It seems the fingers were not strong enough to get through the stem with razor blades. In testing after this, it required extra force from an outside source to get the fingers to fully close. Despite that, the arm was able to bring the peppers to a designated location and drop them off. The basket was able to hold the peppers, but it was not very secure. To end the test the arm scanned past a green pepper and did not attempt to cut that pepper showing the system can successfully tell red from green peppers. Overall the system was a

success with the only issues happening with the hardware. As a team of computer and electrical engineers, this is not ideal, but it worked out okay.

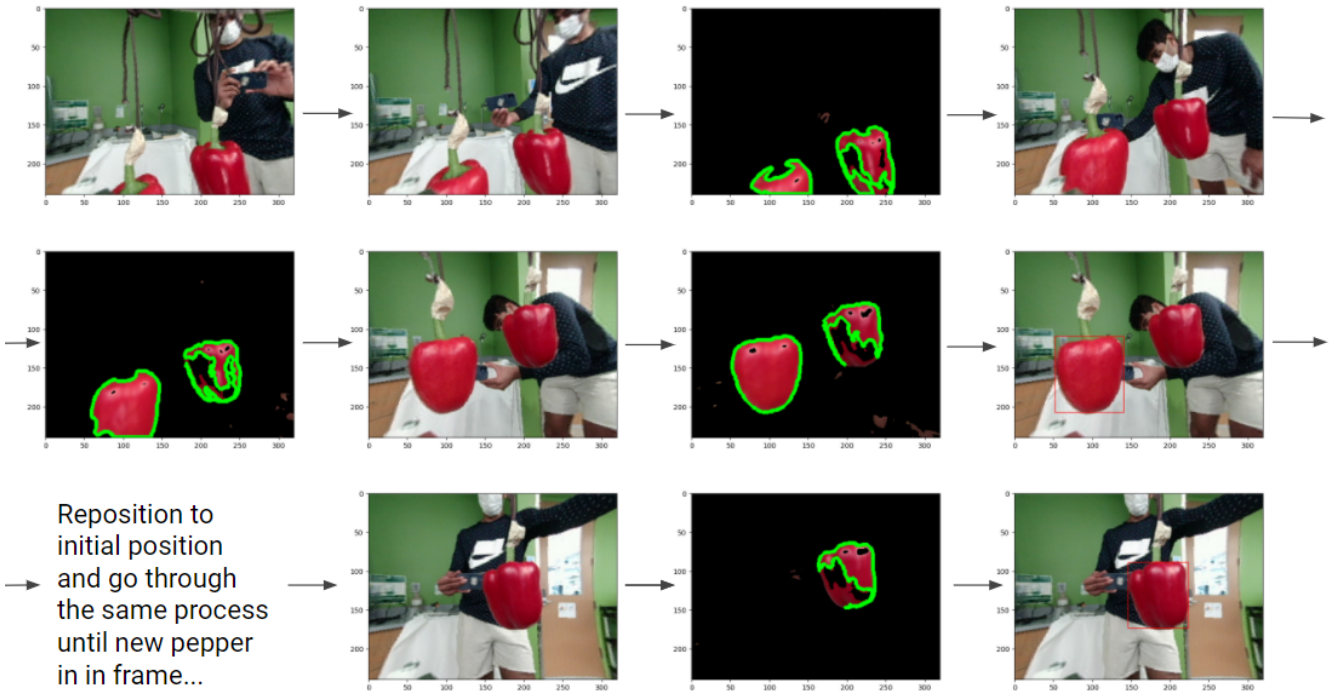


Figure IV.1 Camera Output in real-time Experiment

V. Conclusions

In conclusion, Demeter the harvester successfully accomplished the different requirements for harvesting red-bell peppers, while improving on how accessible a harvester can be. The harvester traversed a row of pepper plants scanning for ripe red-bell peppers, identifying correctly the red-bell peppers to harvest, and ignoring the green peppers. Once it detected a red-bell pepper to harvest, Demeter accurately calculated the depth of the stem and moved to cut it, satisfying the software requirements for the harvester, and the mechanical aspect of navigation where it is directed to. Although Demeter was unable to fully cut a stem, it can be argued that the hardware components were a success as well. Demeter was designed with the intention of being economical, for which reason the hardware was designed to be adapted to the current body to avoid expensive components. The fingers of the robot contain a small inaccuracy when closing (fingers mismatch of over a centimeter), which causes the blades to not align, and ultimately the stem is not completely cut. For the reason that Demeter was designed with the intention of being affordable, these inaccuracies are natural of the design, for which the hardware can be argued to be a success. The hardware should be improved by fully designing an end-factor for Demeter that is more accurate for cutting and collecting peppers.

However, this is a different design goal than what the team intended for Demeter, the reason why it was not developed.

V.I. Next Steps

Even though this project was largely successful, there are some improvements that can be implemented. In the recognition system, the accuracy of the detection can be improved. To do so the first thing that needs to be done is to upgrade the camera. The camera used had a low resolution and from time to time the white balance would not be accurate. This would result in poor color detection and the loss of some features of the scene. Moreover, the intrinsic parameters of the camera were not fully known and some had to be computed. Therefore, the implementation of stereo vision was not very accurate. A camera similar to the Intel RealSense would have likely been a better choice front he start. Finally, if we had mounted a lighting system onto the camera or the arm to maintain consistent lighting when navigating, localizing, and picking we likely could have thresholded and therefore contoured the red candidates better.

On the recognition side we could have used a more intensive contouring algorithm, like the Snake algorithm, that would help us get the best contours without much noise. This could help with being more precise when localizing the pepper candidates and navigating to cut the stem after we have localized a candidate and predicted it was a pepper. Furthermore, one aspect of our recognition and movement algorithm that could hurt performance when the arm is in a field with many half-ripe peppers or false red candidates is the idea that we don't save the locations of ignored (non-picked) candidates. This means that we may encounter them again and try to relocalize on them (a waste of time). Similarly, if we localize on a half ripe pepper candidate that has enough red in the frame, we have no mechanism of determining that it is half-ripe. Therefore the image will be sent to the ML model which will likely predict that the candidate is a pepper (because the model is color agnostic). We could correct this by thresholding for green within the bounding box and if a certain threshold is reached deeming it as unripe and ignorable.

Other next steps could be done improving how the arm driver moves. Setting boundaries with MoveIt for where the plants would be so that it does not plan paths that hit them as the arm moves would be a good first step. Another easy addition would be an ultrasonic sensor on the cutting mechanism that kept the arm a set distance from the plants when scanning. This might not work perfect, but could make the arm more reusable in different scenarios with plants at different distances away. A further easier improvement would be to use action servers in the communication for harvest and reposition. It is better practice and would give more autonomy to the system.

Major next steps could be taken in how the robot harvests the peppers to have a better cut and grip of the peppers. The harvester clearly requires either more torque on the fingers to fully cut the stems, or to have a different cutting mechanism. Same with the net to grip the peppers, they require

more torque. By fully developing the end of the robot instead of creating some fixture to fit the current body, the harvester could correct these current problems.

VI. Budget

960p USB stereo camera --- $82.99
SanDisk 64GB microSD card --- $19.9
Screws #6-32 ⅛” length --- $6.59
Raspberry Pi 4 CPU cooler with case --- $12.9
Raspberry Pi 4 Model B 2019 Quad Core 64 Bit WiFi Bluetooth (4GB) --- $68.34
Razor Blades 1 ½” length --- $17.79

Total --- $208.51

VII. Appendix A

Software Source code

Appendix A.1: Github Demeter: https://github.com/juanpbm/Demeter
Appendix A.2: StereoVision Python library: https://pypi.org/project/StereoVision/
Appendix A.3: Github stereo vision-tutorial: https://github.com/realizator/stereopi-tutorial
Appendix A.4: Contour Methodology:
https://www.sciencedirect.com/science/article/abs/pii/0734189X85900167
Appendix A.5: Github Kinova arm Driver: https://github.com/Kinovarobotics/kinova-ros
Appendix A.6: ILSVRC ImageNet Model: https://www.image-net.org/challenges/LSVRC/