Lab 5 Report

RBE 3001

Conrad Tulig
Robotics Engineering Program
Worcester Polytechnic Institute
Worcester, MA
cctulig@wpi.edu

Kaitlyn Fichtner
Robotics Engineering Program
Worcester Polytechnic Institute
Worcester, MA
kofichtner@wpi.edu

James Ternent
Robotics Engineering Program
Worcester Polytechnic Institute
Worcester, MA
jwternent@wpi.edu

***Abstract*—This lab report contains the methodology for calibrating the camera as well as the implementation and analysis of image processing.**

I. Introduction

The objective of this lab was to implement computer vision to detect and locate objects within the robot's task space. Using a camera positioned above the robot's task space, we applied transformation matrices to convert pixel coordinates to positions in the task space. Then, by applying filters we were able to detect an object's color and size. With the ability to identify objects by color and size, the robot could pick up and sort objects using forward kinematic, inverse kinematic, and inverse differential kinematic trajectory functions developed in previous labs. We also implemented a dynamic tracking function, a URDF simulation of the robot, and detection of an arbitrary object. By integrating computer vision with kinematic control, we were able to attain a more practical use of the robot.

II. Methodology

A. Materials

- RBE3001 Robotic Arm
- Camera
- ST Nucleo-144
- Power Supply
- Ubuntu GNU/Linux Machine with:
  - MATLAB
  - C++ Eclipse

B. Procedure

Camera Calibration

We began by establishing a connection between Matlab and the robot's camera by creating a camera object with **webcam()**. Then we opened the camera calibration app and adjusted the camera's zoom to capture the entire workspace. Once setup, we took 20 pictures with a calibration checkerboard positioned in different configurations. Matlab then identified the checkerboard in each image and determined a set of camera parameters. After checking the histogram of reprojection errors to ensure there were no errors above one pixel, we exported the camera parameters.

Camera-robot Registration

We first calculated the positions of the reference frames for the robot base, checkerboard, and camera. Using these reference frames, we found the transformation matrices between the robot base and checkerboard, the camera and checkerboard, and finally the robot base and camera. With this final transformation, we were able to calculate world points from given XY pixel positions using the **pointsToWorld()** method.

Object detection and classification

Our first step in processing images to identify objects was to calculate the centroids of the colored ball for each object in the task space. To do this, we utilized Matlab's Color Threshold application to generate 3 masks: one for blue balls, green balls, and yellow balls. We exported each respective mask as a Matlab function and applied them separately to the original image in order to isolate each color. We then used the **regionprops()** method to determine the centroids of each colored ball. Next we used the Color Threshold application again to create a mask that filtered for all three ball colors as well as their black base disks and used the **regionprops()** method again to get a list of object areas. Using the positions of each centroid, we determined the corresponding areas for each object.

Automated Sorting System

In order to be able to pick up each object for sorting, we designed a new communication server for controlling the gripper. This server either opened the gripper if the first sent packet was a 1 or closed the gripper for a 0. We then designed the sorting algorithm making use of the object classification from earlier to separate the objects based on their color and size. Finally, culminating together the object detection, inverse differential kinematic trajectories, gripper control, and our sorting algorithm, we implemented a method to identify each

object, navigate to and grab each one, and drop it off at their respective locations.

Dynamic Camera Tracking

Instead of analyzing the work space for objects only once at the beginning, we adjusted our previous sorting process to dynamically track one object in the task space. To do this we repeatedly took a new image of the task space and recalculated the centroid of the target object. This also meant we had to adjust the trajectory's end position to be dynamically changing with the movement of the centroid.

Arbitrary Object Detection

Instead of classifying the provided lab objects designed for this robot to manipulate, we attempted to find and retrieve a differently shaped and colored object. This required recalibrating our image filters to isolate the color of the new object and then implementing our previous trajectory function to grab it.

Robot URDF Model

Our final task for this lab involved replacing the stick model of our robot with a live render of the robot. We started by creating a URDF file that contained the joint and link configurations and parameters as well as the CAD files for each of the robot's links. Then, by importing the URDF file as a MATLAB Rigid Body we were able to display the robot in a figure. We could then update the figure by adjusting the joint angles of the Rigid Body and then rendering the robot again.

III. Results

A. Camera Calibration

After recording 20 configurations of the calibration checkerboard, Matlab was able to successfully analyze each image for the checkerboard and yield each of their positions as shown in *Figure 1*. The mean error from each image was also plotted in *Figure 2*, with a maximum error of around 0.52**.**

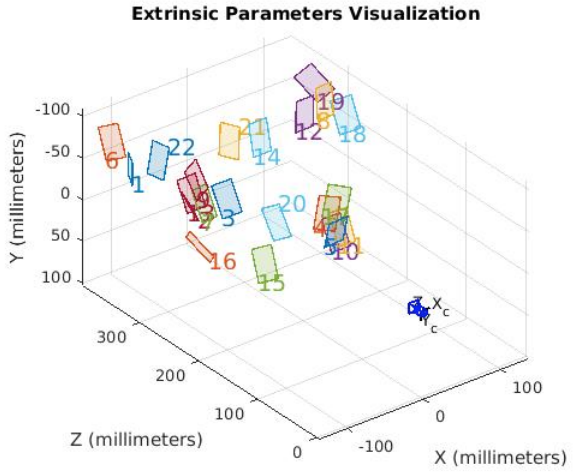

Figure 1: Extrinsic Parameters Visualization

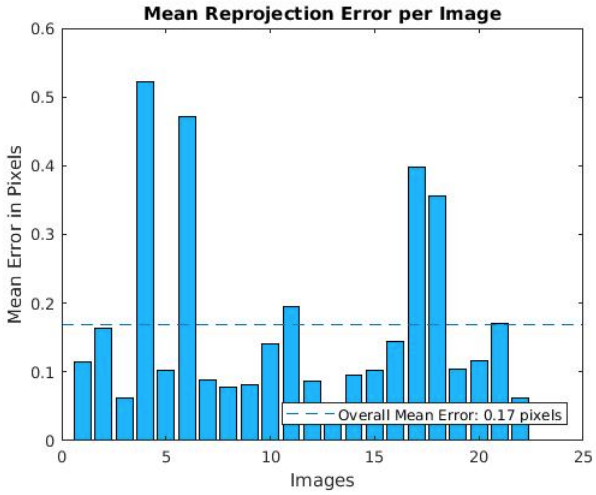

Figure 2: Mean Reprojection Error per Image

B. Camera-Robot Registration

In order to determine the location of objects the camera sees, transformations between multiple reference frames need to be calculated. The transformation from the base of the robot (0, 0, 0) to the reference frame of the checkerboard was experimentally determined through measurements, yielding the matrix in *Table 1*. The transformation from the camera to the checkerboard was determined by Matlab after camera calibrations to be the matrix in *Table 2*. Finally, the transformation from the robot base to the camera was calculated from the previous two transformations to yield the matrix in *Table 3*. Using this transformation matrix from the robot base to the camera, we were then able to convert points in the task in respect to the camera to points in respect to the robot arm. *Table 4* shows the comparison of 4 arbitrary points selected from the camera's view and calculated into world points versus the theoretical positions of these locations.

1	0	0	227.8
0	1	0	101.6
0	0	1	-30
0	0	0	1

Table 1: Transformation Matrix from Robot Base to Checkerboard

0.0266	-0.8675	0.4968	107.7135
0.9995	0.0321	0.0025	101.8156
-0.0181	0.4965	0.8679	290.9126
0	0	0	1

Table 2: Transformation Matrix from Camera to Checkerboard

0.0266	0.9995	-0.0181	128.4418
-0.8674	0.0321	0.4964	47.3483
0.4968	0.0025	0.8678	-336.2277
0	0	0	1

Table 3: Transformation Matrix from Robot base to Camera

World Points		Actual Points	
X	Y	X	Y
153.0818	-7.1131	175	0
198.4145	65.0844	225	76
59.1937	85.3137	73.4	101.6
221.7626	-76.9265	251.2	-76.2

Table 4: Calculated World Points from Pixel Positions versus Actual Theoretical Point Positions

C. Object Localization

Through a series of different filters, we were able to determine both the color and size of each object in the task space. Following the flowchart in *Figure 3*, the first filter applied to the image involved applying a blur filter across the entire image. Then, the 3 different RGB filters were applied separately to the blurred image in order to isolate each object color. After removing some image noise with fill and erode filters, *Figures 9-11* show the resulting black and white filtered image for each color along with the centroid of the object found marked in red. *Figure 4* shows an unfiltered image of all objects in their same respective locations and centroids marked. Finally, in order to calculate the size of each object, a saturation filter was applied to the original image with a broader range of colors to include both the 3 object colors and the black base disks. The result of this filter can be seen in *Figure 5*. The calculated areas, using the centroids of each object, can be seen in *Table 5*.

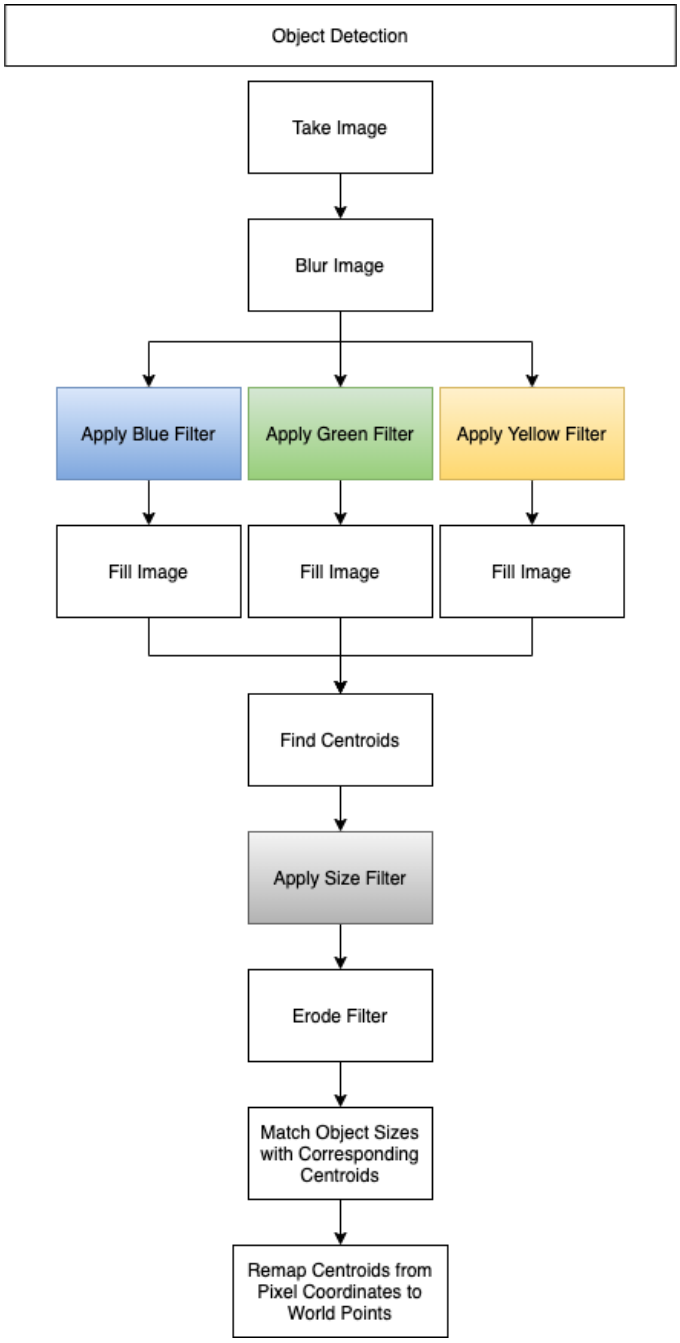

Figure 3: Object Detection Flowchart

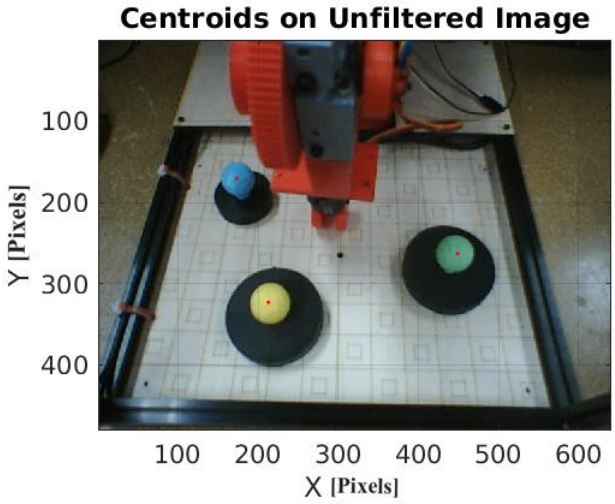

Figure 4: Centroids (in red) on Unfiltered Image

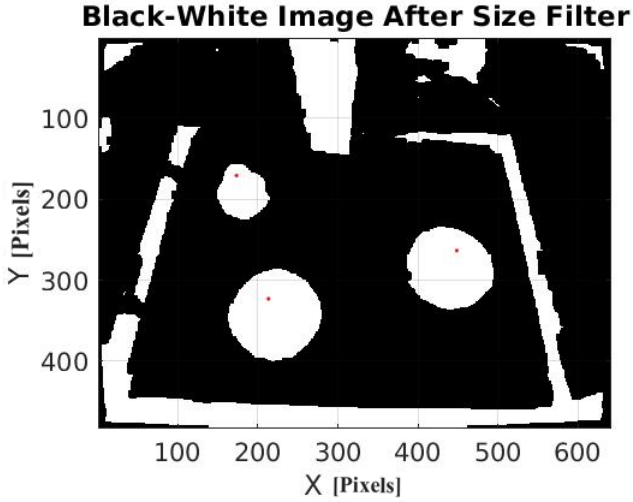

Figure 5: Result of Size Filter with Red Centroids

Color	Centroid X	Centroid Y	Area
Blue	173.4205	170.6202	3215
Green	488.4777	263.3694	8491
Yellow	213.2656	322.9780	9889

Table 5: Centroid Positions and Areas of 3 Differently Colored Objects

D. Automated Object Sorting and Dynamic Tracking

Combining the camera-robot registration and object localization, we were able to find the world point positions of each differently colored and sized objects. Then, navigating to each object using an inverse differential kinematics method, we sorted large objects to the camera's left and small objects to the camera's right. Also, blue objects were put furthest away from the camera, then green, and then yellow closest. A before and after image of this sorting algorithm can be seen in *Figure 6*. A flowchart model of our sorting algorithm from object detection to arm trajectories can be seen in *Figure 12*. A relatively similar process was used to implement dynamic tracking, where the detailed flowchart can be seen in *Figure 13*.

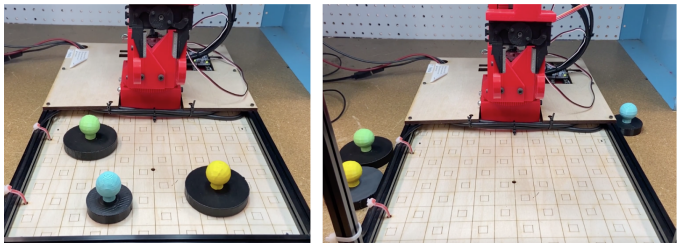

Figure 6: Before Object Sorting (left) and After Object Sorting (right)

E. Arbitrary Object Detection

Once successfully sorting the different colored and sized lab objects, we then attempted to apply the same object detection and retrieval algorithm on an arbitrarily shaped and colored object. We chose to use a dark blue cylindrical drum object as it was both different in shape and color from the lab objects. After adjusting the image filters for the object detection, *Figure 7* demonstrates a successful attempt of the arm finding and retrieving the drum.

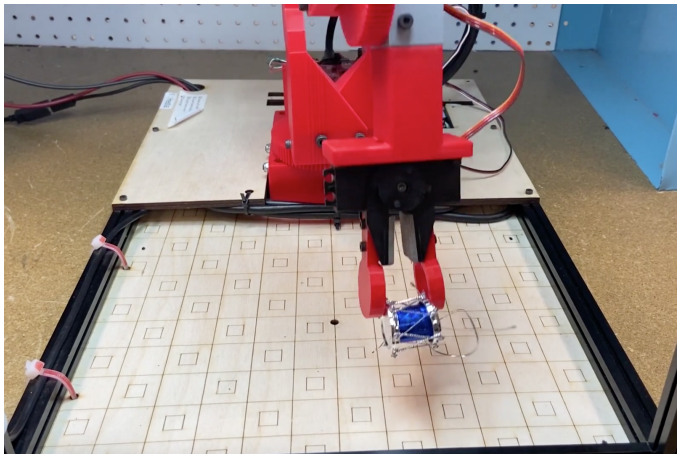

Figure 7: Image of Arm Successfully Identifying and Picking up Arbitrary Object

F. URDF Model

Using the provided CAD models of each robotic arm joint, we attempted to construct a live render of the arm. We were able to get a live render of the arm's reference frames moving with the robot using the URDF file we created. However, the lab station's outdated version of MATLAB was

unable to render the CAD models. As a result we ran the URDF model on our personal machine which unfortunately was not setup to connect to the Nucleo. Instead we simulated a triangular trajectory from a previous lab which can be seen in *Figure 8*.

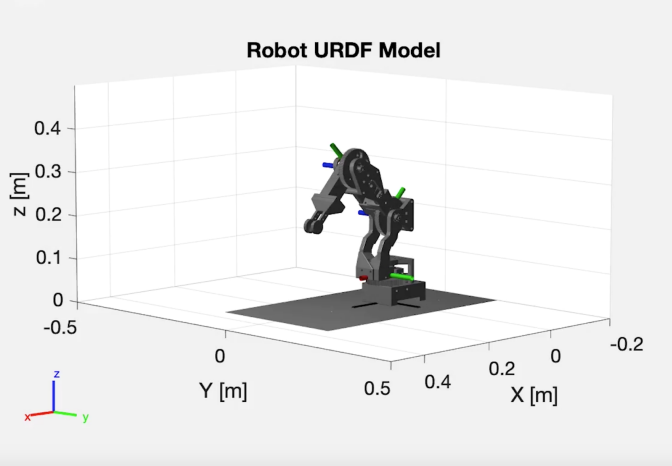

Figure 8: URDF Model of Robot Arm

IV. Discussion

With our camera calibrated, we were able to convert pixel coordinates to locations in the task space. These calculated world points can be seen in *Table 4*. Because the camera parameters could not perfectly translate 2D pixel coordinates into points in our task space, the world points were slightly off from the actual points. We accounted for this inaccuracy by tuning our robot base to checkerboard transformation matrix which can be seen in *Table 1*.

By creating separate RGB filters for each color object, we were able to determine the color of an object. Then, by applying a saturation filter we were able to determine an object's size, utilizing Matlab's **regionprops()** area function. The image filtered for size is shown in *Figure 5*. Since our size filter allowed a broader range of colors, the grey 3D printed part of our robot and the frame around the robot's task space appeared in the filtered image. To return the area of our objects without also including the other areas detected, we cross referenced the centroids of the areas in our size filtered image with the centroids from our color filtered images. If the centroids were within a certain distance of each other, we would return the corresponding area for that object, otherwise that area would be ignored.

While our method consistently detected different colored and sized objects, we found that we could not get an accurate area for an object placed touching the square frame of our task space. Since the frame and the base of the objects are both black, we were unable to filter out the frame and keep the base of the object at the same time. Consequently, if an object was too close to the frame, the frame would become part of the area for that object. Because the centroid of the area including the frame was no longer located at the centroid of the object, our **findObjSize()** function would not return a size for that object. We mitigated this issue by using Matlab's **imerode()** function to shrink the objects' size when applying the size filter so that an objects' area would not connect to the frame in the filtered image. While this method was effective for objects within 1 mm of the frame, it did not always work when an object was directly touching the frame. Since we were still able to detect objects placed within 1 mm of the frame, we decided to just avoid placing objects directly touching the frame for this lab. If an application of this lab required an object to be detected while touching the frame, a solution to this issue could be covering the frame with a different color than our object so it could be filtered out by color. Similarly, if two objects were directly touching each other, the areas could not be calculated separately. Consequently, we decided to always place objects at least 1 mm apart for this lab.

Another issue we discovered was due to the position of the camera. Because the camera is located above the robot and points down towards the task space, there are configurations in which the arm is blocking the camera's view of an object. Regardless of position, with only one camera, the robot will have some configurations in which the arm is blocking the camera's field of view. To ensure we could get a clear image of the task space, we had the robot start in a configuration with the end effector lifted 80 mm above the task space. We then used an image taken while the robot was in its starting configuration to determine how to sort the objects. This method was effective except for when dynamic tracking. Since we were constantly obtaining new images in dynamic tracking, it was possible for the object to be blocked by the arm in certain configurations. However, it was rarely an issue since the robot tracked in an up position and only moved down towards an object that was no longer moving.

When our dynamic tracking algorithm did not detect an object, the arm would continue to move towards the last object detected. This allowed the arm to grab an object despite the object becoming out of view of the camera as the arm approached the object. Consequently, if an object was moved out of view of the camera, the robot would attempt to pick up an object in the most recent position detected by the camera. If the object was not moved back into view of the camera before the robot reached this position, the program would stop dynamic tracking.

Localizing and picking up an arbitrary object presented additional challenges because the geometry of the object needed to be compatible with the robot's gripper. We chose a drum Christmas tree ornament because it was about 1 cm smaller in width than the open gripper and had tall enough sides for the gripper to grasp. The drum can be seen in the robot's grasp in *Figure 7*. We found that while in the correct orientation, the gripper could easily grab the drum, picking up the drum from its narrower side was much more challenging. This issue was expected as the gripper was designed for picking up the round objects provided to us in the lab.

To enhance the stick model of the robot used in prior labs, we implemented a URDF model of the robot, as seen in *Figure 8*. The URDF model was much more useful for

modeling how the robot would interact with the task space, since it shows the actual geometry of the arm. For a more complicated application, a URDF model could be very useful for simulating a scenario where the arm needs to avoid obstacles in its trajectory planning. However, the URDF model is limited in that it is very detailed and consequently slow to run and unable to update in real time.

V. Conclusion

By integrating the use of a camera into our robotic system, we were able to implement controls developed in previous labs for the practical application of picking up and sorting objects. Utilizing an inverse differential kinematic approach, we were able to navigate towards the objects for both sorting and dynamic tracking. Creating different filters for size and color allowed us to experiment with Matlab's image functions and image processing tools. While working with arbitrary objects, we also learned the importance of designing a gripper for its specified task. Furthermore, creating a URDF model of the robot taught us a useful way to simulate the robot's movement within the task space. Overall, this project taught us about many of the challenges to consider in robotic manipulation as well as strategies using modeling and simulation to discover problems before running code on the robot.

Video of Robot: https://youtu.be/K5jmX0ql7Wc

VI. Appendix

Appendix A: Authorship

Section:	Author
Introduction:	**Kaitlyn Fichtner**
Methodology:	**Conrad Tulig/James Ternent**
Results:	**Conrad Tulig**
Discussion:	**Kaitlyn Fichtner**
Conclusion:	**Kaitlyn Fichtner**

Appendix B: Black-White Image After Color Filters

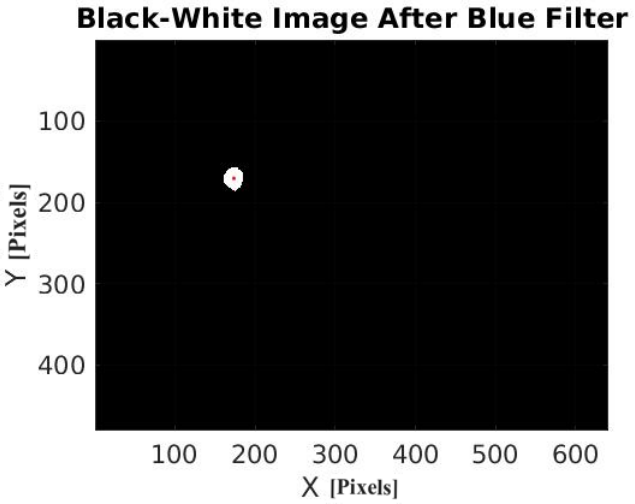

Figure 9: Result of Blue Filter with Red Centroid

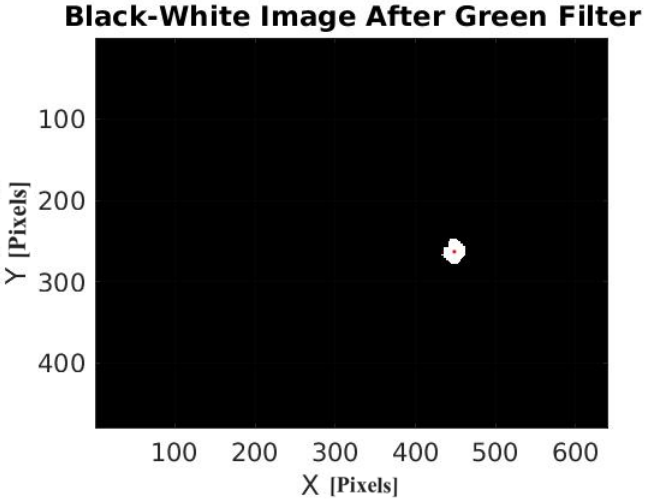

Figure 10: Result of Green Filter with Red Centroid

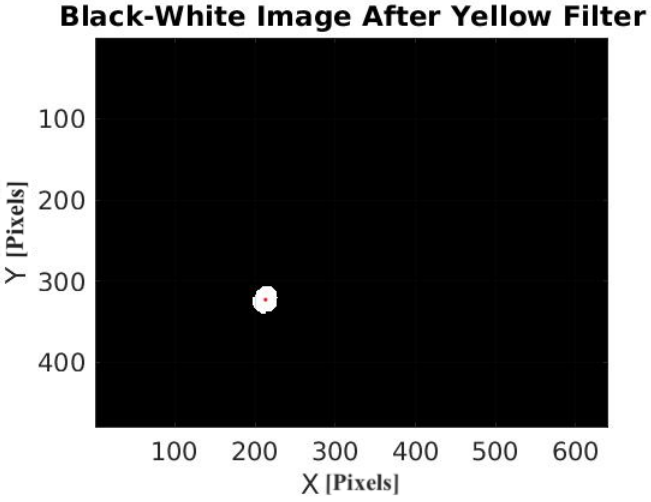

Figure 11: Result of Yellow Filter with Red Centroid

Appendix C: Detailed Flowcharts

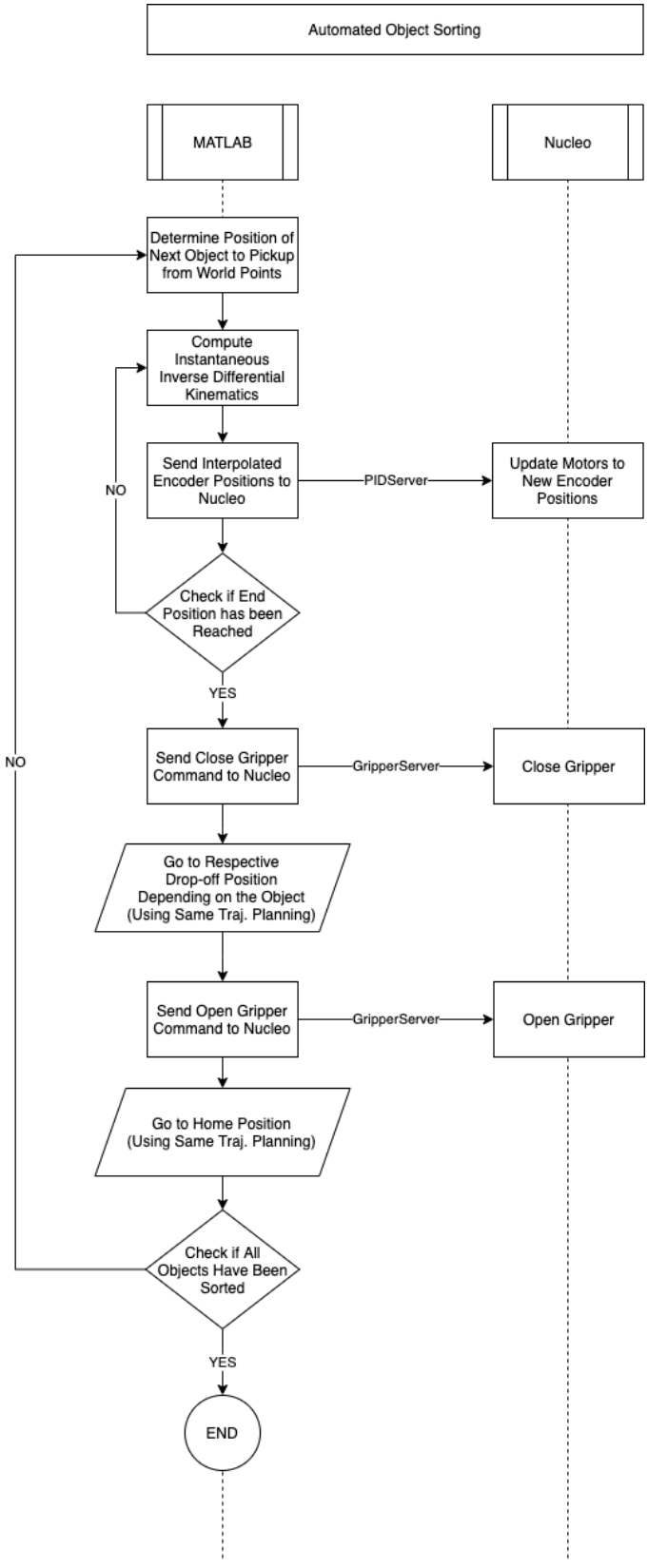

Figure 12: Automated Object Sorting Flowchart

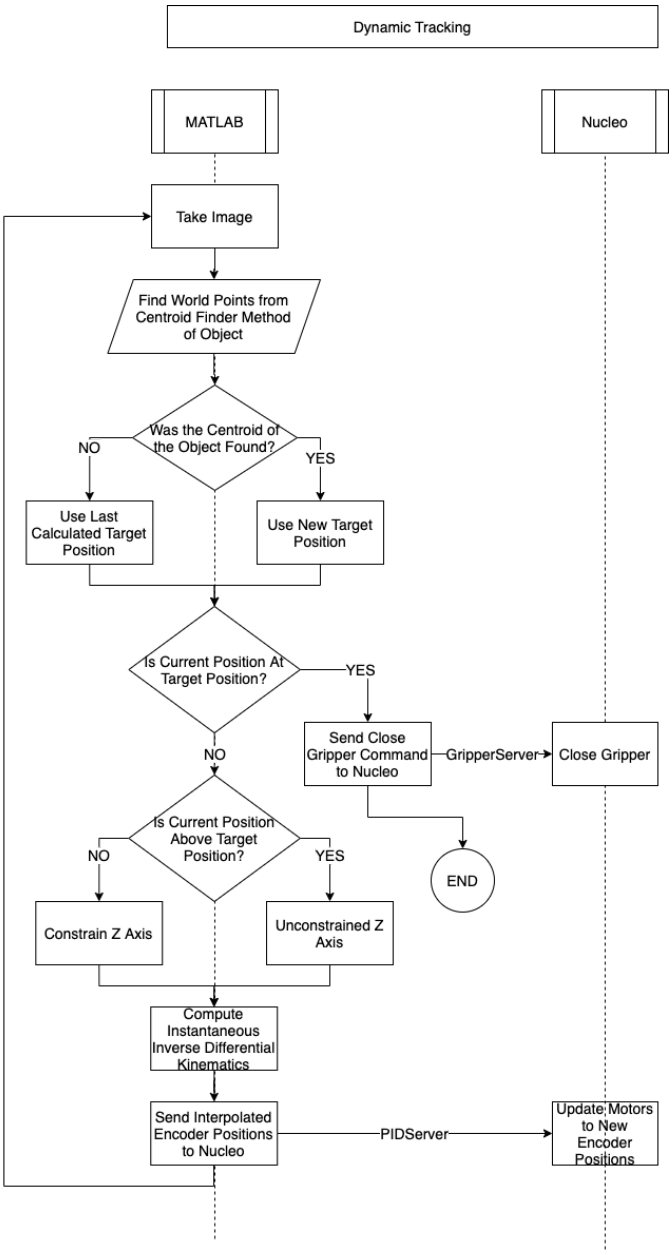

Figure 13: Dynamic Tracking Flowchart