HUMAN-INSPIRED ROBOTIC HAND-EYE COORDINATION

by

Stephanie T. Olson

A Thesis Submitted to the Faculty of

College of Engineering and Computer Science

In Partial Fulfillment of the Requirements for the Degree of

Master of Science

Florida Atlantic University

Boca Raton, FL

August 2018

HUMAN-INSPIRED ROBOTIC HAND-EYE COORDINATION

by

Stephanie T. Olson

This thesis was prepared under the direction of the candidate's thesis advisor, Dr. Erik Engeberg, Department of Ocean and Mechanical Engineering, and has been approved by the members of her supervisory committee. It was submitted to the faculty of the College of Engineering and Computer Science and was accepted in partial fulfillment of the requirements for the degree of Master of Science.

SUPERVISORY COMMITTEE:

[illegible]

Erik Engeberg, Ph.D.
Thesis Advisor

[illegible]

Javad Hashemi, Ph.D.

Oren Masory

Oren Masory, Ph.D.

[illegible]

Javad Hashemi, Ph.D.
Chair, Department of Ocean and Mechanical Engineering

[illegible]

Stella N. Batalama, Ph.D.
Dean, College of Engineering and Computer Science

Khaled Sobhan

Khaled Sobhan, Ph.D.
Interim Dean, Graduate College

August 2, 2018

Date

ACKNOWLEDGEMENTS

The author wishes to express sincere gratitude to her advisor Dr. Engeberg for his guidance on and funding of this project. This work was supported by the Department of Energy Minority Serving Institution Partnership Program (MSIPP) managed by the Savanah River National Laboratory under SRNS contract TOA#0000332969 in collaboration with Florida International University’s Applied Research Center and Idaho National laboratory. This research was also supported by the NIH: NIBIB award # 1R01EB025819 and I-SENSE at FAU. Thanks to Dr. Hashemi and Dr. Masory for being on the supervisory committee as well as everyone in the BioRobotics lab for their encouragement and assistance throughout the project.

ABSTRACT

Author:	Stephanie T. Olson
Title:	Human-Inspired Robotic Hand-Eye Coordination
Institution:	Florida Atlantic University
Thesis Advisor:	Dr. Erik Engeberg
Degree:	Master of Science
Year:	2018

My thesis covers the design and fabrication of novel humanoid robotic eyes and the process of interfacing them with the industry robot, Baxter. The mechanism can reach a maximum saccade velocity comparable to that of human eyes. Unlike current robotic eye designs, these eyes have independent left-right and up-down gaze movements achieved using a servo and DC motor, respectively. A potentiometer and rotary encoder enable closed-loop control. An Arduino board and motor driver control the assembly. The motor requires a 12V power source, and all other components are powered through the Arduino from a PC.

Hand-eye coordination research influenced how the eyes were programmed to move relative to Baxter's grippers. Different modes were coded to adjust eye movement based on the durability of what Baxter is handling. Tests were performed on a component level as well as on the full assembly to prove functionality.

DEDICATION

This manuscript is dedicated to my Mom, Dad, and sister, Danielle for their love and support.

HUMAN-INSPIRED ROBOTIC HAND-EYE COORDINATION

TABLES

FIGURES

EQUATIONS

1 INTRODUCTION

1.1 Goals and Applications

The overall goal was to design and fabricate novel humanoid robotic eyes and program them to move logically with a Baxter robot based on hand-eye coordination research. Two control goals were position and temporal control. Position control in which gaze accurately followed a Baxter end effector was sought first followed by temporal control that resulted in human-like eye movement with respect to time.

The design and its capabilities have the potential to benefit any industry that employs human-robot interaction. Assembly line robots in manufacturing (like Baxter), assistive robots in the service industry, and audio-animatronics for entertainment in theme parks are just a few examples.

1.2 Literature Review

The research done falls under five categories: the Baxter robot, the human eye, current robotic eye designs, realism and robots, and hand-eye coordination.

1.2.1. Baxter Robot

The Baxter Robot from Rethink Robotics is depicted in Fig. 1. Baxter has two arms each with seven degrees of freedom and a maximum reach of 1210mm. Two types of grippers are available to act as Baxter’s hands: vacuum cup grippers and electric parallel grippers. There is a camera in each arm and force sensors in each joint. The mobile pedestal consists of four caster wheels to move him and four anchors to secure him in place. Baxter is mainly used in manufacturing settings, packaging, loading and unloading items from assembly lines among other tasks.

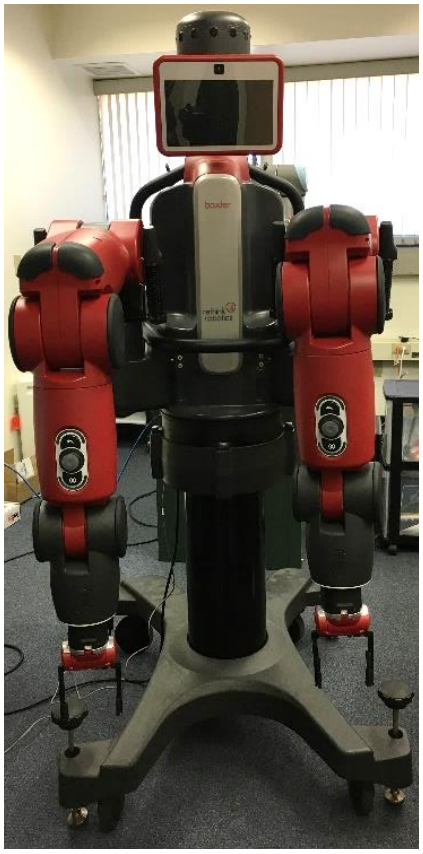


Fig. 1. Baxter robot

Unlike many robots, Baxter is not programmed but trained. Once Baxter is enabled, the user can freely move either of Baxter's arms while holding onto his respective wrist. An arm holds its orientation when the wrist is released. For a user who wants to do as little coding as possible, they can run the available joint recorder Python file. While the user moves the arms, the code samples Baxter's position at a specified sampling rate and writes the data to a file. For Baxter to mimic his training movements, a second Python file called joint trajectory file playback sends commands back to Baxter by accessing the data file written by the recorder. A slightly more hands-on approach to training Baxter is also applied in this paper.

Baxter's "face" is a monitor, and the eyes on his display are limited to two vertical positions, forward and down, and three horizontal positions, left, right, and center.

1.2.2. The Human Eye

The average diameter of a human eye is 24mm [1], [2]. Contrary to how they are often modeled, eyeballs are not perfect spheres; the cornea's radius of curvature is about 4mm smaller than that of the overall eyeball [1]. Pupil diameter can range from 3mm to 7mm [1].

An eye's mechanical range of motion is referred to as the oculomotor range, or OMR [3]. The OMR of a human can be defined by how far left, right, up and down the average person can rotate their eyes. Different sources define the human OMR using slightly different numbers, ranging from 53° ± 2° to 60° for both left and right eye rotation [2]–[4]. However, it is agreed that quick eye movements, which are known as saccades, are neurally limited to the effective oculomotor range (EOMR) of ±45° horizontally [3], [4]. Ninety-percent of the time, human eyes move within the range of ±20° [5]. There is also a discrepancy between sources as to the vertical limits of the EOMR; one source claimed it is 20° up and 29° down while another claimed it is closer to 35° up and 47° down [2], [4].

The larger a saccade, the larger the saccade's duration, peak velocity and average velocity [6]. The maximum velocity of a saccade is around 600°/s [3], [6]–[8]. Saccade durations range between approximately 25 and 100ms for saccades larger than one-degree [6], [7], [9], and they take about 200ms to initiate [2].

1.2.3. Current Robotic Eyes

Various existing robotic eye designs were analyzed. Every design researched simplified the eyeball as a perfect sphere. The Agile Eye was used as a base model in multiple journal papers [2], [5], [10]. The original design consists of three DC servo motors mounted to the vertices of a triangular plate [11]. The motors are mounted at an angle with their shafts pointing up towards the center of the plate. Two spherical links connect each DC motor to the eye, giving a total of six spherical links.

The speed and range of the Agile Eye are greater than that of a human's; however, the mechanism does not fit well behind an artificial human face. To remedy this, one group modified the design just over ten years ago utilizing bent links that allowed the servos to be reoriented farther behind the eye [5]. Although the capabilities of both designs are impressive, their complexity makes them bulky.

More recently, a paper on a bipedal robot named Romeo was released [2]. The robot contained simplified Agile Eyes that had two activated degrees of freedom while roll was passive. Although simpler, the eyes' range of motion was greatly reduced.

In a different mechanical approach, a cable/tendon system was used for robotic eyes in a face meant to simulate a human having eye surgery [12]. The system was achieved using servos and wires. The group's initial design consisted of two servo motors controlling both eyes simultaneously and relied on wires that would stretch and regain shape with actuation. A lack of tension in certain wires lead to replacing the connections with aluminum wire, but the rigidity of the new wire made the old design obsolete. To accommodate, the final design consisted of two servos for each eye. An issue with servo-wire mechanisms that was not discussed is the interdependence of the servo motors; the movement of one motor could rotate an eyeball in a way that causes the second motor to stall.

A group called Inmoov designed robotic eyes made of 3D printed components and two servos as shown in Fig. 2 [13]. The top servo rotates the eyes left and right by moving a long piece connecting both eyes. The bottom servo moves both the eyes and top servo for vertical eye rotation. The low profile, few actuators, and independence of horizontal and vertical eye movement all make this design appealing. One potential drawback is lack of speed; the videos of the assembly in motion all have the eyes rotating relatively slowly [14]. One foreseeable reason is that running the eyes faster could cause the bottom servo's elbow connection to jam.

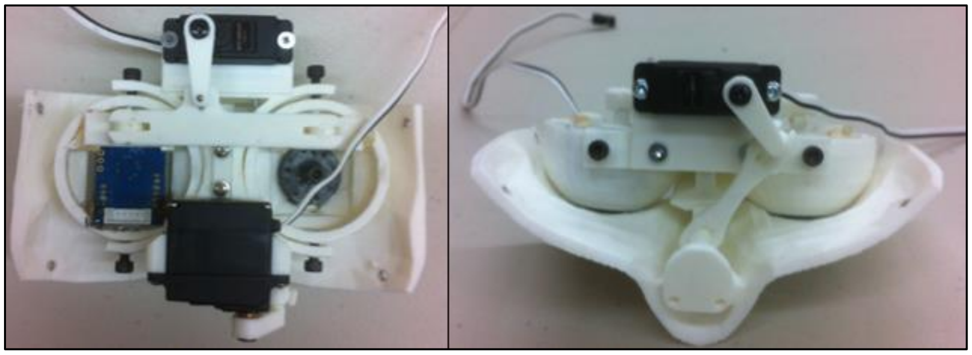


Fig. 2. InMoov eye mechanism back (left) and bottom (right) [14]

Some recent robotic eyes do not utilize any motors for actuation. In one design, an artificial eye is suspended in fluid within a translucent outer shell [15]. An eight-coil electromagnetic drive structure attached to the outer shell is used to rotate the eyeball. The structure consists of pairs of adjacent coils on the top, bottom, left, and right of the outer shell with one coil from each pair mounted closer to the pupil and one closer to the back of the eyeball. Small currents are sent to specified coils that cause the electromagnets to either push or pull corresponding fixed magnets, altering the orientation of the eyeball within its shell.

The design's small footprint and ease of installation make it practical for animatronics; the original eyeball mechanisms in older animatronics can be removed and replaced with the new design with little effect on the rest of the robot. The eyes can move at a maximum velocity of 500°/s, which is comparable to that of a human eye. One design limitation is accuracy. Because the eyes move through open-loop control, the eyes can overshoot the desired position.

Another actuator-free robotic eye design is display eyes. For instance, Disney Enterprises, Inc. has a patent out for robotic eyes that are simply curved OLED screens mounted to the inside of animatronics [16]. The concept of display features on animatronics has already been implemented in some of the rides at Walt Disney World to bring classic animated characters to life. Disney Research Pittsburgh also came up with their own display eyes made from printed optics instead of screens [17]. Because display eyes glow by their nature, they are limited to only appearing "real" for animated characters.

1.2.4. Realism and Robots

In the previous section, a lack of realism was said to be a negative design aspect of robotic eyes. To justify that claim, the following needed to be answered: how does realism affect how a robot is perceived, and how does that perception affect the robot's performance?

A study done by the MIT Media Lab sought to answer both [18]. First, a test was conducted using three "characters": one robotic, one animated, and one human. Only the eyes of each character were visible to test subjects through rectangular openings. The subjects were given instructions by each character and then rated their perceptions of each on a questionnaire. The results showed that the robot was perceived as more credible, informative, and engaging than the animated character. The enjoyability of the interaction was also greater for the robot than the animated character. In all categories, the robot's ratings were comparable to those of the human. As it applies to this report, a Baxter robot's interaction with humans would improve in the same categories if it used robotic eyes in place of display screen eyes.

A person trusting in something that looks and moves like they do is an instinct that can be observed in other animals. One study analyzed the behavior of actual guppies around a robotic guppy [19]. RoboFish, as it was called, was placed in a tank of guppies, once without its glass eyes and once with. The amount of time guppies spent with the eyeless RoboFish was significantly less than that spent with the RoboFish with eyes. Also, the guppies spent the same amount of time with other live guppies as they did with the RoboFish with eyes. Like in the MIT study, the robot's results were comparable those of the real thing.

The group performed a second test that involved the RoboFish's movement. It moved in a natural zig-zag pattern at varied speeds for one trial and at a constant speed in a straight line for another. The results: the live guppies spent significantly more time with the naturally moving RoboFish. The journal paper emphasized the importance of realism in both appearance and movement of robotics in interaction-related applications.

Going back to human analyses, one study focused on how the movement of robotic eyes affected the persuasive power of a robot as a storyteller [20]. The results proved that the use of gaze increased a robot's persuasiveness. Oddly, when the robot used gestures and NOT gaze, the robot's persuasiveness diminished. Although gaze and gesture had clear impacts on persuasion, neither had a statistically significant impact on the user's perception of the robot (i.e. likeability, perceived intelligence). The question arises whether the use of gaze would affect human perception if the robot performed a task other than storytelling.

The benefit of gaze in human-robot interactions was found in another study in which a robot handed an object to test subjects [21]. The results showed that the handoff time was shorter when the robot used human-like gaze than when it did not use gaze. In short, the efficiency of the task increased with the use of gaze.

A recent study went in depth to analyze the effects of three different types of gaze: averted, situational, and constant [22]. The study had three sample groups, one to test each gaze type. Subjects would view a game on a monitor in which an object was placed under one of three cups. The cups would be shuffled, and subjects would have to guess which cup contained the object. The game had three difficulty levels based on shuffling speed. An Aldebaran Nao humanoid robot would play along with the subject employing one of the three types of gaze. It would agree or disagree with the subject's answers, sometimes doing the latter even when the subject was correct. Subjects had the option to change their answer if the robot disagreed with them.

Gender played an unexpected role in the results. While females were least likely to change their answers when the robot employed constant gaze and most likely to do so when the robot employed situational gaze, the opposite was true for males. Due to the studies small sample size of male subjects, further testing would be required to make definitive conclusions. Regardless of gaze, subjects tended to trust the robot more when their initial guess was incorrect, suggesting uncertainty in answers increased trust.

1.2.5. Hand-Eye Coordination

In a study done by Umeå University in Sweden and Queen's University in Canada, eye gaze was tracked while test subjects performed a task. The task was to grab a bar, picked it up, and moved it so the tip of the bar touched a target above the grasp site [7]. There were three setups: one free of obstacles, one with a square obstacle and one with a triangular obstacle. Areas or objects of interest in the workspace were referred to as landmarks, such as the grasp site, obstacle, and target.

Multiple conclusions were gleaned from the experiment. First, gaze always led hand movement, typically leading by one-second or less. Additionally, gaze exited a landmark around the time of a kinematic event at that landmark. An example of a kinematic event would be when the bar contacted the target. The landmarks were divided into two categories. The grasp site, target, and support surface were labeled as obligatory gaze landmarks. The tip of the bar and parts of the obstacles that stuck out were deemed optional gaze landmarks since fewer and shorter gaze fixations were associated with them. Interestingly, the eyes never fixated on the hand or the object being moved.

Another task-based hand-eye coordination study analyzed gaze while test subjects made tea [23]. Body movement, gaze, and object manipulation were charted on a single timeline. Gaze was typically directed to an object one-second or less before contacting the object; however, lead times as large as two-seconds did occur. A separate study was performed for a sandwich-making task that yielded similar results [24].

The head authors of both studies cowrote a paper comparing the two experiments [25]. For consistency, they defined an object-related action (ORA) as uninterrupted single-object manipulation. The example given of one ORA was picking up an object, moving it, and setting it down. They went on to compare saccade sizes within and between actions from both studies. The distributions of within-action saccades for both tests were very similar; both had averages of about eight degrees and maximums less than 40°. There was a large difference in the maximum between-action saccade sizes: 90° for tea-making versus 30° for sandwich-making. The difference was credited to the fact that the tea-making task required moving around a room while the sandwich-making task was done sitting down.

It was found that task difficulty effected the frequency of gaze fixations on landmarks. One study conducted an object manipulation test in which subjects had to drag and drop emails into folders [26]. The test was divided into two trials: easy and difficult. In the easy task, explicit directions were given to organize the emails. The difficult task required reading the email and judging which folder it belonged in; i.e. if the email had to do with travel, the subject would decide to put it in the “travel” folder. On average, subjects spent 45% of the time looking at landmarks (referred to as areas of interest) during the easy task versus 62% during the difficult task. After additional testing beyond object manipulation, the study concluded that the amount of time a subject fixates on an area of interest is directly related to area’s complexity.

A similar study was done that found a relation between gaze fixations and the difficulty of a driving task [27]. The more difficult the driving task, the more time subjects spent looking at the road and the more concentrated their gaze fixations were at the center of the road. The results are consistent with those of the object manipulation test.

By applying the research, the question, “Where will Baxter look?” was answered. Baxter’s preplanned trajectory had him reach for an object, pick it up, move it, and set it down. Regardless of what Baxter was handling, the eyes would look at the grasp site until the object was grabbed. While moving the object, future points along the known trajectory would be fixated while periodically moving gaze to the target location. The target would be defined as the object’s final desired location. The proximity of those future points to the gripper’s current position and the frequency of target fixations would depend on the difficulty associated with handling a given object. The difficulty would be an input argument in the code.

The maximum saccade speed of the robotic eyes was decided based on a data set of 1316 measured saccades that included saccade amplitude and velocity. The set is from the previously referenced study by Umeå University and Queen’s University [7]. Although the data agreed with the other studies referenced that a human eye’s maximum saccade speed was about 600°/s, the data showed that few saccades occur at velocities over 500°/s. Therefore, a maximum saccade speed of 500°/s was chosen as a design requirement for the robotic eyes.

The chosen horizontal range of motion for the robotic eyes was ±35°, which is 10° shy in either direction of the average human EOMR but well beyond the ±20° range where human eyes operate ninety-percent of the time. Because Baxter usually performs tasks in which his lower and especially upper peripherals are rarely used, the vertical range of motion was chosen to be ±20°.

2 DESIGN

The robotic eye design is divided into three subassemblies: the eye mechanism, wiring, and the hardware to mount the assembly to Baxter's display screen.

2.1 Robotic Eyes

The two robotic eyes were designed to move in tandem. In other words, the eyes look in the same direction and cannot cross. The benefit of the design not seen in current robotic eyes is independent vertical and horizontal movements that do not sacrifice speed or accuracy of gaze fixations.

2.1.1 Version 1

Fig. 3 is the basis of the design: the eyes. The spherical eyes are 15/32 inches (about 24mm) in diameter, comparable to the size of human eyes. A two-piece shell secures each eye, acting as a socket for the sphere. L-brackets connect the shells to a four-by-six-inch plate, and standoff blocks are used to adjust the distance between the eyes and plate. Each eyeball has a cylindrical extrusion with an eyebolt screwed into the circular face. Eyebolts screwed into either end of a cylindrical bar are lined up with the those connected to the eyes. Shoulder screws were placed through the eyebolts and secured with nuts to form pin connections between the bar and eyes.

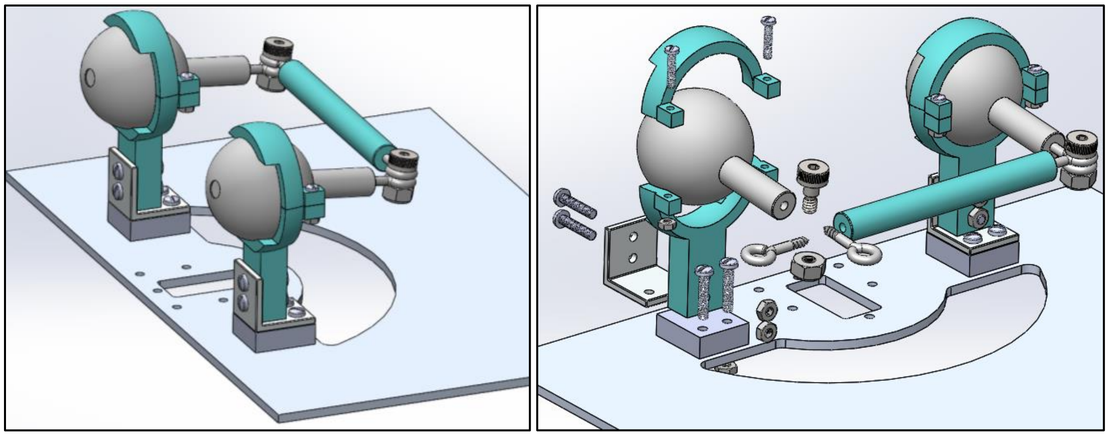


Fig. 3. Basis of eye mechanism assembled (left); partially exploded view (right)

The part of the assembly in Fig. 4A enables the eyes to look up and down. The bar connecting the eyes goes through a slot that is part of the DC motor housing. Two set screw shaft collars keep the motor housing at the center of the bar. The bar is free to move along the length of the slot as the eyes rotate to “look” up and down, as shown in Fig. 4B. The DC motor and rotary encoder shaft are fixed to either end of a threaded rod (which is simply a screw with its head removed) by set screw shaft couplers, as shown in Fig. 4C. The encoder outputs the direction and number of DC motor rotations and is used for closed-loop motor control. The track along which the encoder slides up and down is attached to extrusions from a hollow disk that sits on top of the plate. The rod is threaded through a press-in nut fixed into the encoder track. When the motor shaft rotates, it threads the rod through the press-in nut, causing the motor housing and encoder to move up or down depending on whether the shaft is rotating counterclockwise or clockwise, respectively.

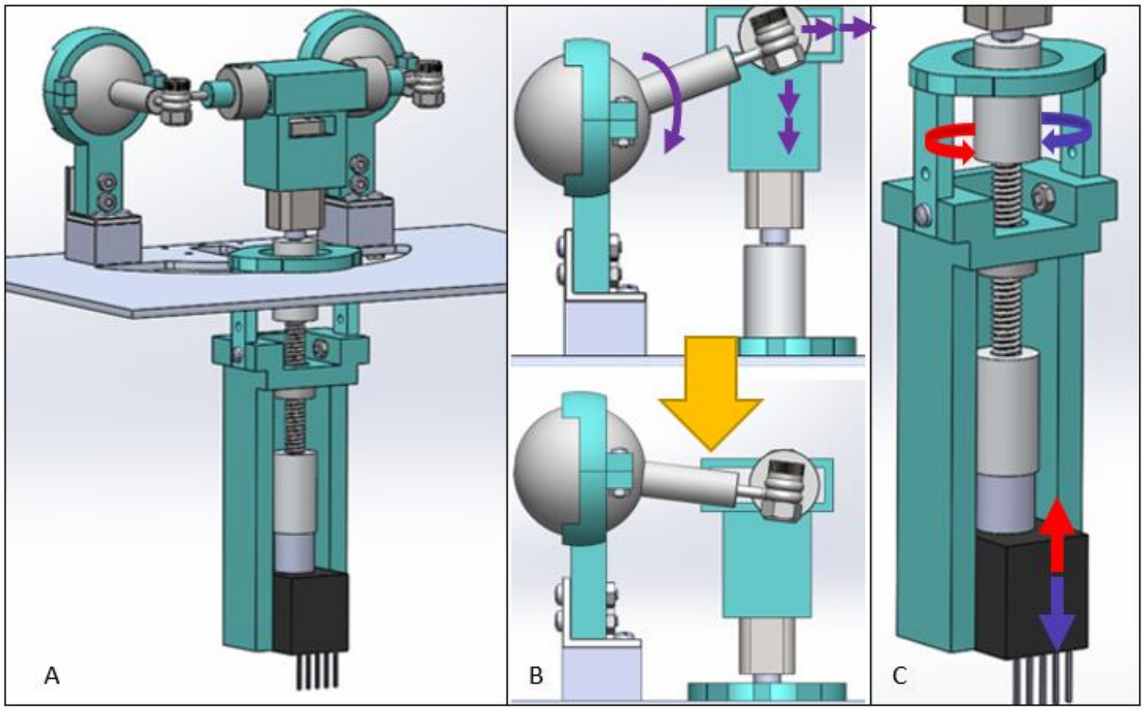


Fig. 4. Motor-encoder assembly for vertical eye movement

When the bar slides back and forth within the DC motor housing slot, the lever arm of the vertical force acting on the bar changes. The bar’s displacement within the slot and, consequently the lever arm, does not change linearly. Because the eye is rotating, the bar’s centroid moves along an arced path in the XZ plane. Therefore, the part of the mechanism that allows the eyes to look up and down is a nonlinear dynamic system.

The servo in Fig. 5A enables the eyes to look left and right. The servo arm is connected to the hollow disk extrusions above the encoder track. The servo acts like a windshield wiper, sliding the motor-encoder assembly along the plate within the arced cutout shown in Fig. 5B. As mentioned previously, the eyes were designed to move 35° left and right. They are kept within this range by the connection points used to assemble the shells; however, the range could easily be expanded by making shells that had connection points on the sides instead of in the back. The depicted design was kept for the convenience of having two reference points at which the approximate servo angle was known.

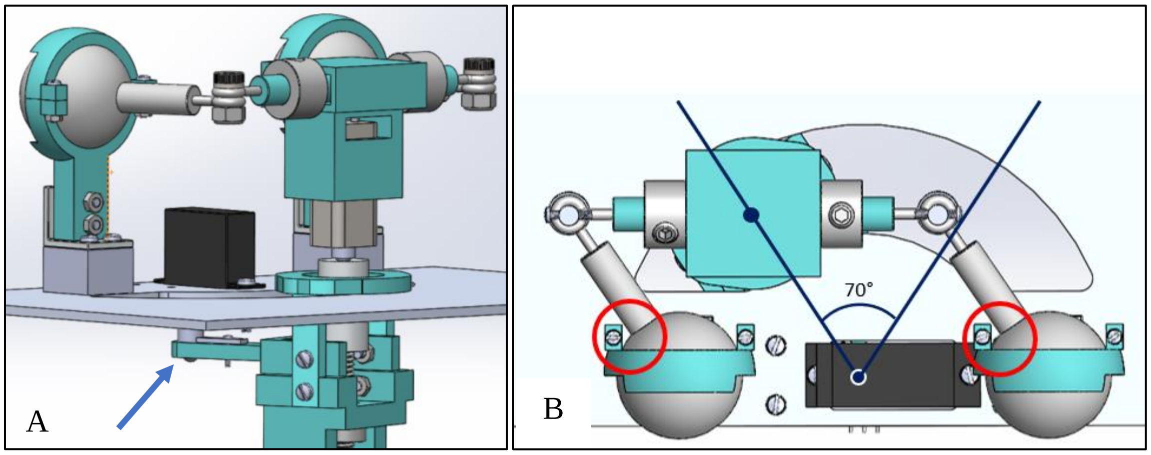


Fig. 5. Servo assembly for horizontal eye movement (left); top view of servo range of motion (right)

To measure the servo's position and enable closed-loop servo control, a potentiometer is mounted below the plate as shown in Fig. 6. The boxed piece in the figure acts as an interface between the potentiometer shaft and the servo arm.

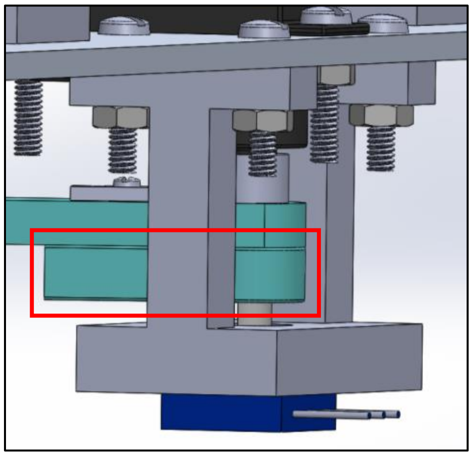


Fig. 6. Servo-potentiometer interface

2.1.2 Final Design

A few design changes were made to Version 1. Although the initial servo motor was sized based on the expected load of the assembly, it proved incapable of overcoming the friction between the hollow disk and the plate. Therefore, a servo capable of producing a greater torque was chosen. The stronger servo was larger, which meant its orientation on the plate needed to rotate ninety degrees as shown in Fig. 7. The change led to the second modification: elongating the potentiometer mount.

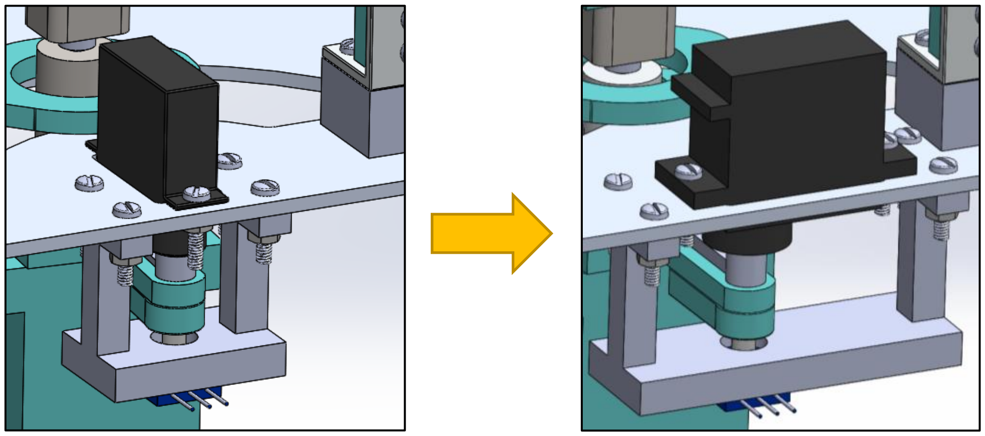


Fig. 7. Version 1 (left) versus final (right) servo-potentiometer orientation

Due to the constant contact between the eyebolts, shoulder screws and nuts during left and right eye movement, the nuts would loosen and eventually fall off the assembly. The shoulder screws and nuts were replaced with clevis pins shown in Fig. 8 that used small screws instead of the usual wire for a more reliable connection.

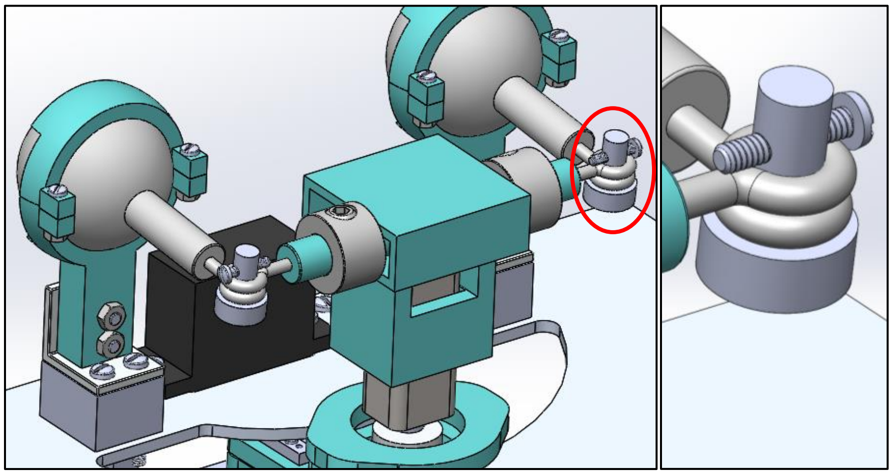


Fig. 8. Final pin connections

The final modification was the additional support piece in Fig. 9. The addition ensured that the hollow disk remained parallel to the plate without raising on the left or right during movement. To prevent the addition from being so close to the plate that it hindered servo movement, it was attached to the hollow disk extrusions via slots (red), allowing the distance between it and the plate to be adjusted. Slots also replaced holes in the servo arm and encoder track (blue) to ensure the servo arm was parallel to the plate.

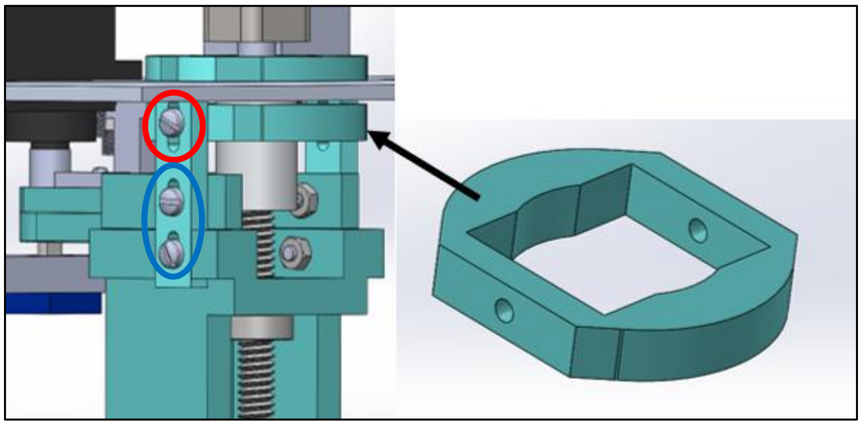


Fig. 9. Final design replaced certain holes with slots and added support piece

The rectangular plate from Version 1 was susceptible to warping when removed from the 3D printer build plate, as shown in Fig. 10A. Therefore, the thickness was increased for the final plate shown in Fig. 10B. In addition, the front corners of the plate were chamfered so the assembly could fit behind an artificial face. The length was increased to make room for four holes to connect the plate to the Baxter mount.

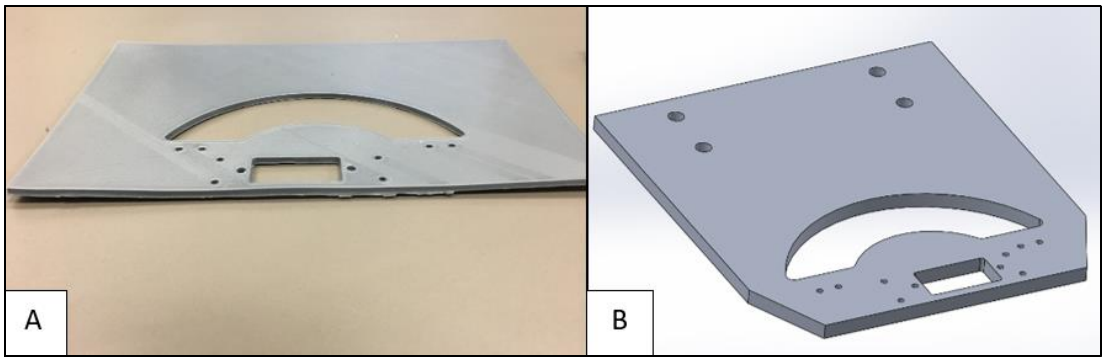


Fig. 10. Original plate warped (left); final plate design (right)

2.1.3 Customizing

In future applications, the distance, D, between the eyes may need to be altered, especially if the eyes ever need to fit behind a preexisting artificial face. To modify the assembly given the value of D as a design requirement, only the distance between the sets of holes used to anchor the eyes to the plate and the distance between bar eyebolts shown in Fig. 11 need to be adjusted.

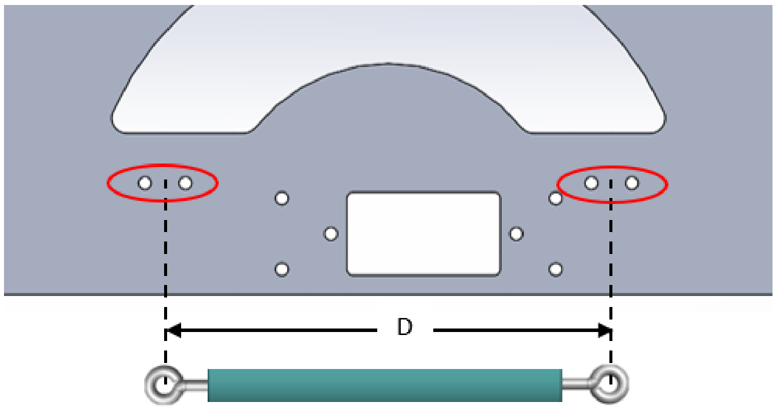


Fig. 11. Components dependent on the distance between eyeball centroids

The distance the eyes are from the front of the plate can limit how close the eyes can be installed behind a face. Fortunately, the eyes can be placed closer to or farther from the front edge of the plate by moving the same holes previously mentioned. Fig. 12 shows that this change only affects where the bar sits inside the DC motor housing slot. The only time the assembly would need to be changed further is if the eyes were placed so far forward or back that the length of the housing slot would need to be increased.

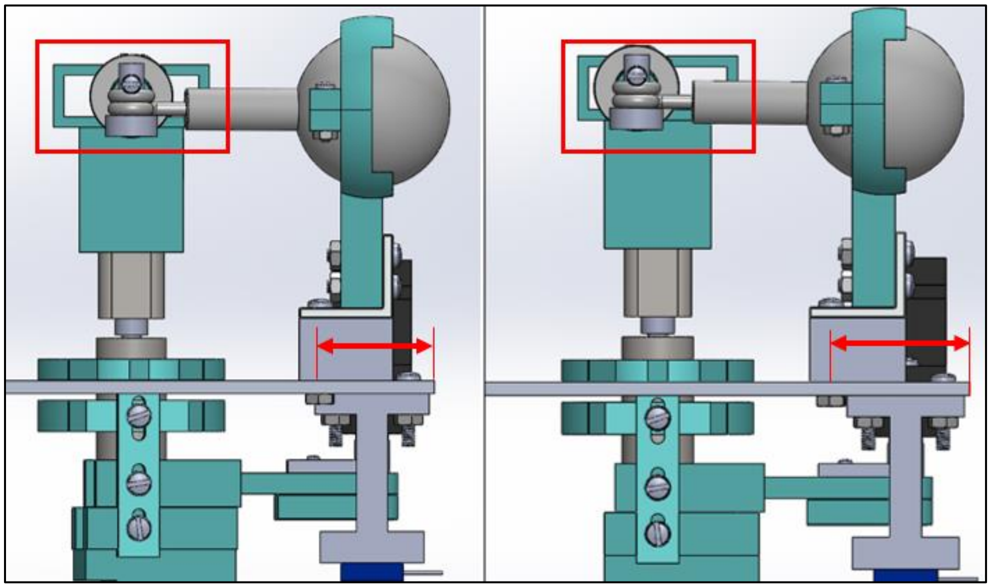


Fig. 12. Moving the eyes forward or back only effects the bar’s location in the slot

In place of the single-piece eyeballs, two-piece eyeballs shown in Fig. 13 were designed to house laser diode modules. The modules act as miniature laser pointers, and each consists of a small cylindrical laser housing with an APC driver circuit attached to the back with two lead wires. The laser housing sits in the eyeball, and the lead wires thread out of the elliptical hole in the stem. The eyeball pieces are press-fit together and cannot twist once secured. The assembly’s range of motion is slightly reduced using the two-piece eyes because the circular ridge where the pieces meet cannot slide under the shell.

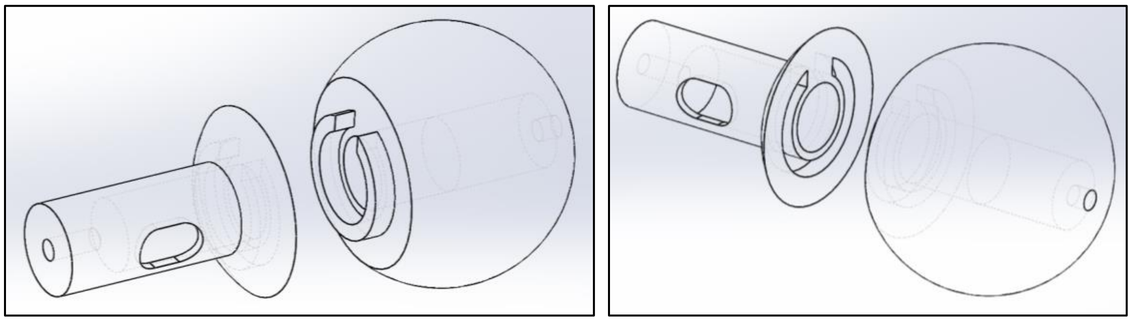


Fig. 13. Two-piece eyeball to house laser diode module

2.1.4 Sizing Parts

As previously mentioned, the servo was sized based on the length of the servo arm and weight of the attached load. The torque was calculated using Equation 1 where g is acceleration due to gravity (9.81m/s^2), m is mass in kilograms and l is length in centimeters. The masses of screws and nuts were neglected.

$$\tau = g\left(m_{couplers} + m_{threaded\,rod} + m_{DC} + m_{encoder} + m_{collars} + m_{plastic}\right)l_{arm} \qquad (1)$$

The full calculation in Appendix A resulted in a required torque of 2.47N·cm. A plastic-gear servo with a maximum torque of 7.85N·cm was chosen, giving a factor of safety of 3.18. Unfortunately, the servo was not reliable and often stalled. A slightly larger motor with metal gears and a stall torque of 17.7N·cm was chosen to ensure functionality and durability. The old and new servo are shown in Fig. 14.

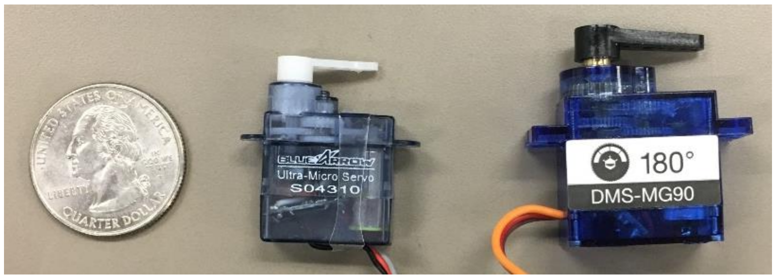


Fig. 14: (from left to right) a quarter, the original servo, and the final servo

The parts required for vertical bar displacement were sized next. In the calculations, the bar was approximated to be level with the eyeball eyebolts even though the former sits on top of the latter. First, the maximum tangential velocity, $V_{t,max}$, shown in Fig. 15 was calculated from the maximum saccade speed chosen for the design, 500°/s, and the eyes' radius of rotation. The maximum vertical velocity of the bar, $\vec{V}_{y,max}$, was found to be equivalent to $V_{t,max}$ in the y-direction.

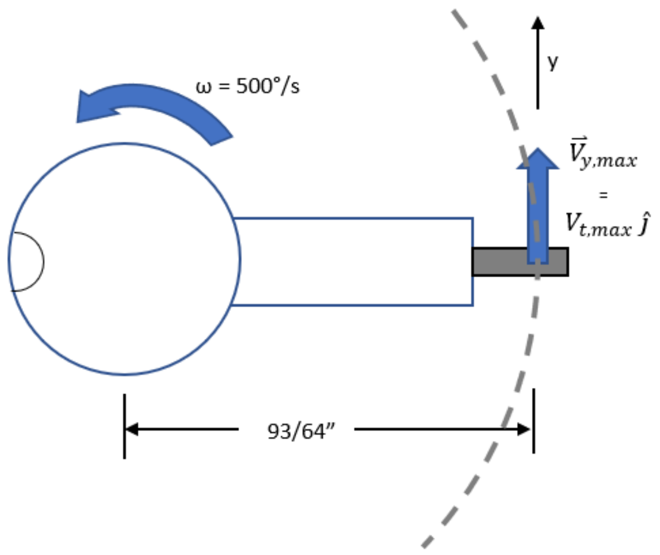


Fig. 15. Maximum angular velocity of eyeball, moment arm, and vertical velocity of bar

Next, two parameters needed to be determined simultaneously: the DC motor’s maximum angular speed, $\omega_{DC,max}$, and the threaded rod’s threads-per-inch. The finer the thread, the faster the motor would need to rotate to achieve $V_{y,max}$. Table 1 shows the values of $\omega_{DC,max}$ calculated for three of the UNC screw sizes analyzed.

Size	Threads Per Inch	Pitch, P (in)	Desired Eye ω (°/s)	Desired Eye ω (rev/s)	$V_{t,\,max}$ (in/s)	DC motor RPM, $\omega_{DC,max}$
#8	32	0.03125	500	1.389	2.018	3875.0
#10	24	0.04167	500	1.389	2.018	2906.3
¼”	20	0.05000	500	1.389	2.018	2421.9

Table 1. Calculated RPM required for each screw size to achieve specified maximum saccade velocity

All three calculated RPMs could be reached with a small DC motor; however, a 10-24 screw would only require seventy-five percent of the motor speed needed for an 8-32 screw, ruling out the latter. The 10-24 was chosen over the ¼” because the decrease in required RPM was not large enough to justify the increase in screw diameter. The detailed velocity, threads-per-inch and RPM calculations are in Appendix B.

After choosing the 10-24 screw, a DC motor capable of rotating at 2907RPM was needed. To get a high RPM from a motor small enough to fit in the assembly, a DC motor with a 5:1 gear ratio was selected as shown in Fig. 16. The motor cost less than $15 and can rotate at 2500RPM when supplied 6V and 4900RPM when supplied 12V, well over the 2907RPM required.

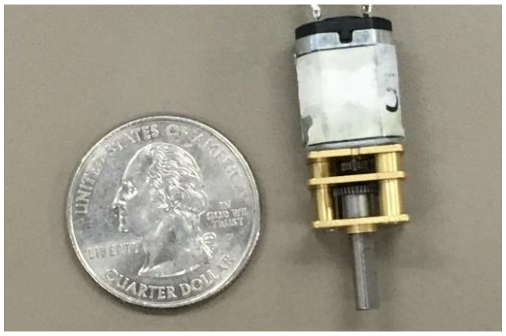


Fig. 16. Geared DC motor selected next to quarter

2.2 Wiring

An Arduino Uno R3 controls the servo, potentiometer, and rotary encoder as shown in Fig. 17. The board is powered by the PC running Arduino through a USB connection. The encoder requires a 220Ω resistor, and the servo and potentiometer are each connected to a100uF capacitor. The capacitors, while not required to run the assembly, smooth out voltage dips in the board such as when the servo motor begins to rotate [28]. The laser diode modules mentioned in the customization section of the design could also be powered through the Arduino board. Each module would require a 100Ω resistor in series.

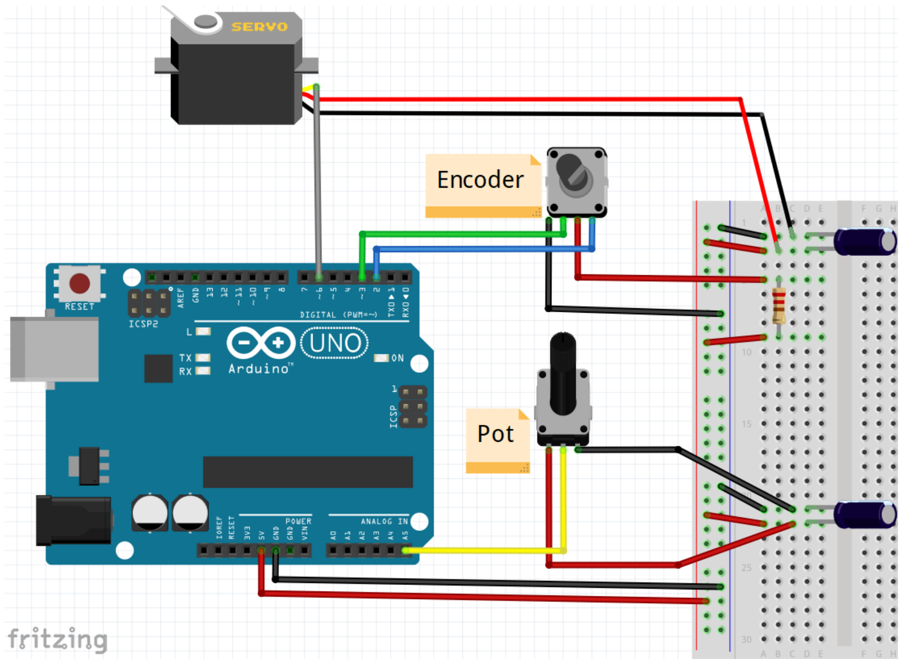


Fig. 17. Wiring diagram for servo and both sensors

A Pololu Dual MC33926 motor driver shield for Arduino in Fig. 18 controls the DC motor. The shield can be inserted directly onto the Arduino board, which saves space.

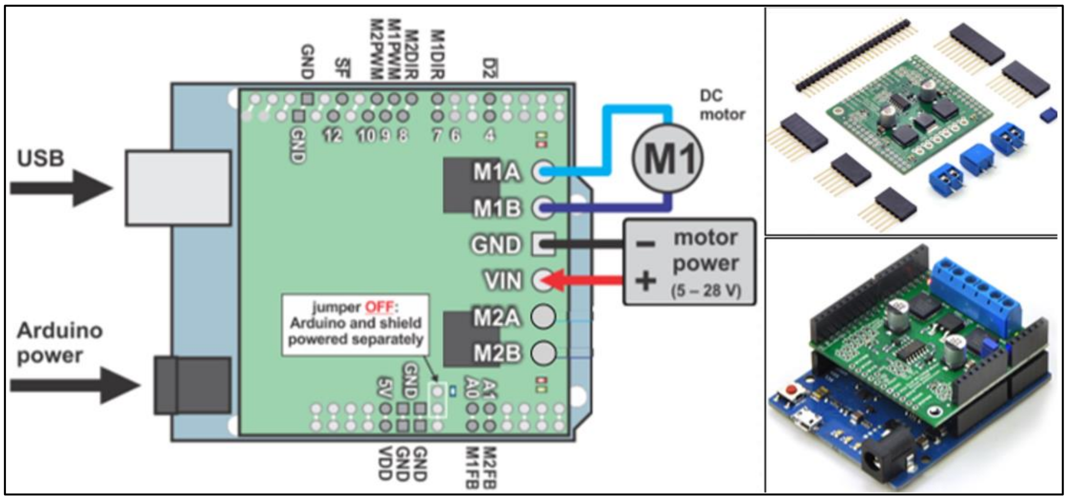


Fig. 18. Pololu motor driver: diagram (left), parts (top-right), on an Arduino board (bottom-right) [29]

Because the DC motor requires a 12V power supply to reach the calculated 2907RPM, it cannot be powered by the Arduino board. For testing, the motor was powered by the BK Precision 1672 variable power supply shown in Fig. 19. The power supply ensured a constant 12V was delivered to the motor, which a draining battery could not guarantee. A maximum current draw lower than the motor's specified stall current was also set to protect the motor. Both aspects made the variable power supply ideal for testing; however, the supply's capabilities far exceed what is necessary to run the motor. The extra capabilities also make it far more expensive than a battery. In short, a 12V battery would be a better choice to power the DC motor outside of a testing environment.

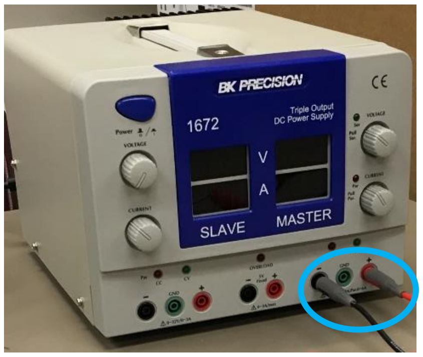


Fig. 19. BK Precision 1672 variable power supply utilizing one power and one ground connection

2.3 Baxter Mount

The mounting hardware attached to the assembly is shown in Fig. 20. Two L-brackets connect the plate to the two mounting supports shown in red. The mounting supports mirror one another and fit over the top corners of Baxter's display monitor. A slim bracket and two sets of screws and nuts can be used to align the mounting supports; however, because the supports are designed to fit tightly on Baxter's monitor, this bracket is optional. On the plate, one L-bracket is secured using two short screws and nuts. Longer screws are used on the left side of the plate along with spacers to elevate a small plate shown in purple above the main plate. The small plate is for mounting the breadboard. Because Baxter's monitor is tilted down towards the ground, the angle of each L-bracket needed to be less than ninety degrees for the assembly to be level when mounted.

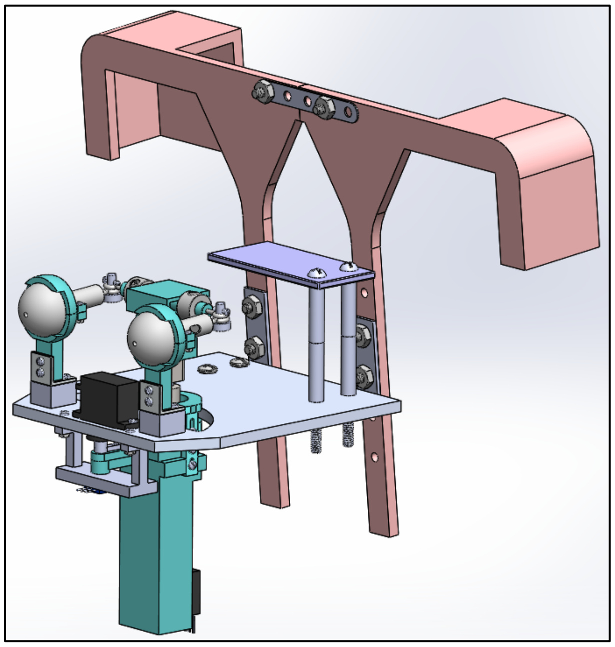


Fig. 20. Isometric front views of attached Baxter mount and breadboard mounting plate

To prevent the left mounting support from contacting the breadboard plate, the latter was designed with a notch as shown in Fig. 21.

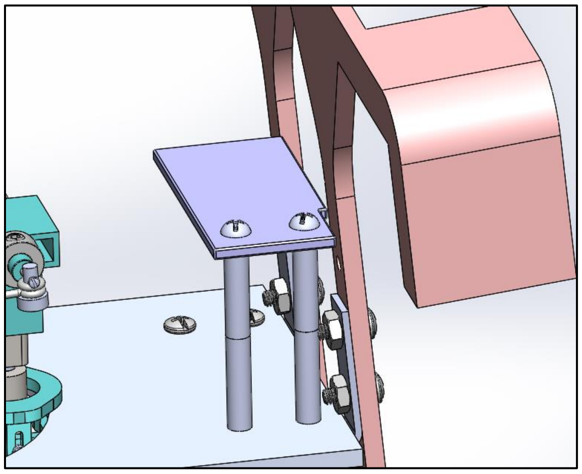


Fig. 21. Breadboard plate

The view of the assembly in Fig. 22 shows the six screws attached to the mounting pieces with the screwheads in the back to reduce contact with Baxter’s display screen. Countersinks were later made for the six holes to further mitigate screen contact.

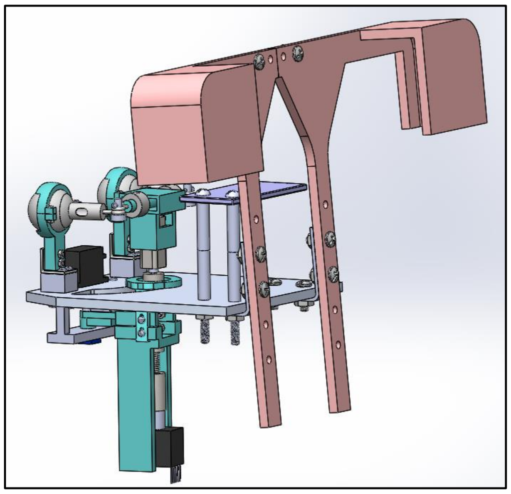


Fig. 22. Isometric back view of attached Baxter mount and breadboard mounting plate

A mask was created in CAD using a 3D scan of a human face. Grooves in the back of the mask shown in Fig. 23 guide where the robotic assembly sits behind it. A band made from Velcro (not shown) wraps around Baxter’s display screen and connects to the mask at the ears. The tension in the band can be adjusted by separating one of the Velcro connections at an ear, sliding more Velcro through and reestablishing the connection.

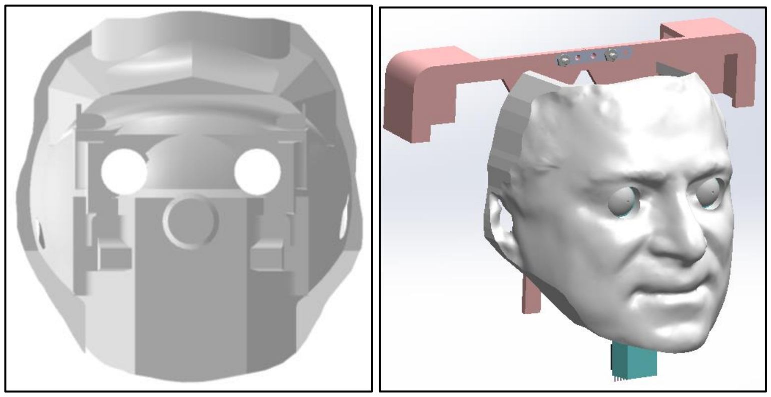


Fig. 23. Back of mask (left); mounted assembly with mask (right)

3 SOFTWARE

Fig. 24 shows the communication between all components. Baxter's desired poses are input into the PC through ROS. The gripper location and the percent open the gripper is are obtained by the Arduino. The Arduino outputs to the servo and motor driver and receives the output of both sensors. The motor driver sends the Arduino's speed commands to the DC motor. Arduino publishes coordinates such as gripper position and actual gaze fixation point to ROS topics that are written to text files on the laptop.

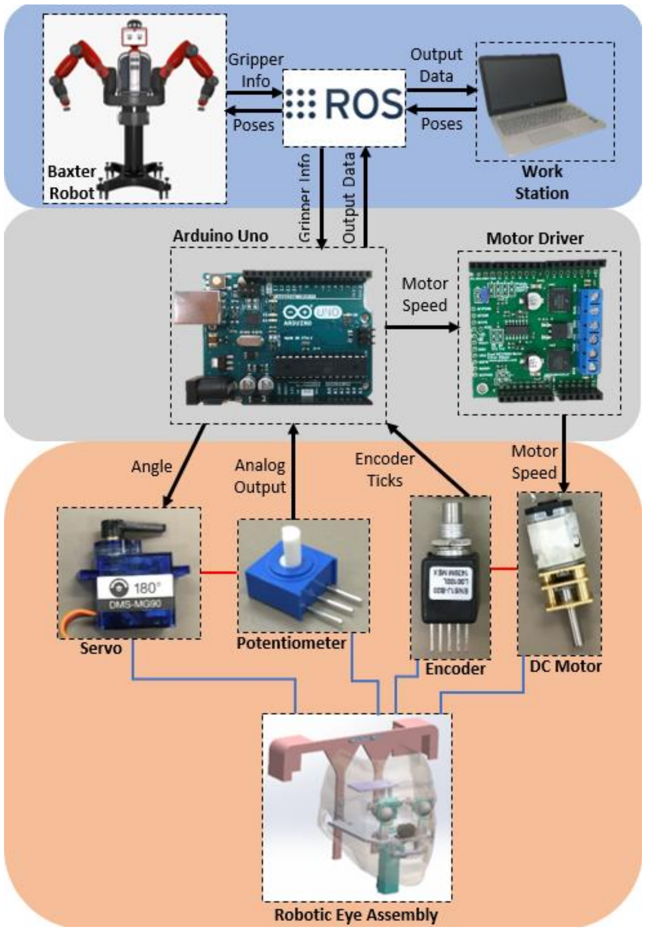


Fig. 24. Communication between components

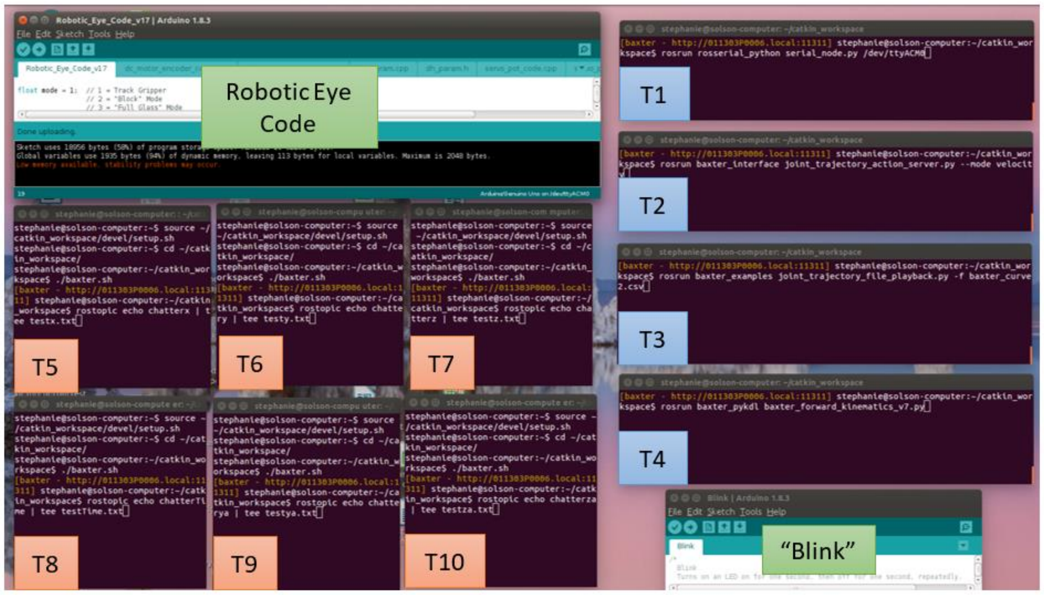
3.1 System Requirements

A laptop was used to communicate with Baxter and execute all necessary code. Baxter's system requirements as well as his workstation setup instructions were found on the Rethink Robotics website [30]. The required Linux distribution to run Baxter, Ubuntu 14.04, was installed on the laptop. His recommended programming language, ROS Indigo, was also installed. An ethernet cable was used to establish communication between the laptop and Baxter.

Additional requirements to run the eye assembly were Arduino software and Python 2.7. For communication from the laptop to the Arduino Uno, the Uno was connected to one the laptop's USB ports.

3.2 What to Run

The following needs to be open on the laptop before running the eyes with Baxter: the Arduino robotic eye code, the Arduino "Blink" sketch, and between three and ten Linux terminals depending on how Baxter is asked to move and if data is acquired for analysis. The setup is shown in Fig. 25. Each terminal needs to source the baxter.sh file to establish a ROS environment. In the first terminal, Baxter's motors are enabled. To then run the eyes with Baxter, the Arduino sketch is uploaded to the Arduino Uno. Once the upload is complete, serial communication between ROS and Arduino is established in the same terminal in which the motors were enabled. The second and third terminals are used to command Baxter's movements, which is detailed in the next section. The fourth terminal runs Python code used to publish the left gripper's x, y, and z location as well as the degree to which the left gripper is open to ROS topics. Terminals five through ten are optional for data acquisition, echoing ROS topics published to by the Arduino code and exporting the data to text files. To use those terminals, all six echo commands need to be executed before the Arduino code is uploaded. To stop the Arduino code when finished, the serial communication in the first terminal is terminated and the “Blink” sketch is uploaded to the Uno.



```
Robotic_Eye_Code_v17 | Arduino 1.8.3
float mode = 1;  // 1 = Track Gripper
                 // 2 = "Block" Mode
                 // 3 = "Full Glass" Mode
Done uploading.
Sketch uses 18956 bytes (58%) of program stora
Global variables use 1935 bytes (94%) of dynamic memory, leaving 113 bytes for local variables. Maximum is 2048 bytes.
Low memory available, stability problems may occur.
```

```
[baxter - http://011303P0006.local:11311] stephanie@solson-computer:~/catkin_wor
kspace$ rosrun rosserial_python serial_node.py /dev/ttyACM0
```

```
[baxter - http://011303P0006.local:11311] stephanie@solson-computer:~/catkin_wor
kspace$ rosrun baxter_interface joint_trajectory_action_server.py --mode velocit
y
```

```
[baxter - http://011303P0006.local:11311] stephanie@solson-computer:~/catkin_wor
kspace$ rosrun baxter_examples joint_trajectory_file_playback.py -f baxter_curve
2.csv
```

```
[baxter - http://011303P0006.local:11311] stephanie@solson-computer:~/catkin_wor
kspace$ rosrun baxter_pykdl baxter_forward_kinematics_v7.py
```

```
stephanie@solson-computer:~$ source ~/
catkin_workspace/devel/setup.sh
stephanie@solson-computer:~$ cd ~/catk
in_workspace/
stephanie@solson-computer:~/catkin_wor
kspace$ ./baxter.sh
[baxter - http://011303P0006.local:113
11] stephanie@solson-computer:~/catkin
_workspace$ rostopic echo chatterx | t
ee testx.txt
```

```
stephanie@solson-computer:~$ source
~/catkin_workspace/devel/setup.sh
stephanie@solson-computer:~$ cd ~/ca
tkin_workspace/
stephanie@solson-computer:~/catkin_w
orkspace$ ./baxter.sh
[baxter - http://011303P0006.local:1
1311] stephanie@solson-computer:~/ca
tkin_workspace$ rostopic echo chatte
ry | tee testy.txt
```

```
stephanie@solson-computer:~$ source
 ~/catkin_workspace/devel/setup.sh
stephanie@solson-computer:~$ cd ~/c
atkin_workspace/
stephanie@solson-computer:~/catkin_
workspace$ ./baxter.sh
[baxter - http://011303P0006.local:
11311] stephanie@solson-computer:~/
catkin_workspace$ rostopic echo cha
tterz | tee testz.txt
```

```
stephanie@solson-computer:~$ source ~
/catkin_workspace/devel/setup.sh
stephanie@solson-computer:~$ cd ~/cat
kin_workspace/
stephanie@solson-computer:~/catkin_wo
rkspace$ ./baxter.sh
[baxter - http://011303P0006.local:11
311] stephanie@solson-computer:~/catk
in_workspace$ rostopic echo chatterTi
me | tee testTime.txt
```

```
stephanie@solson-computer:~$ source
~/catkin_workspace/devel/setup.sh
stephanie@solson-computer:~$ cd ~/ca
tkin_workspace/
stephanie@solson-computer:~/catkin_w
orkspace$ ./baxter.sh
[baxter - http://011303P0006.local:1
1311] stephanie@solson-computer:~/ca
tkin_workspace$ rostopic echo chatte
rya | tee testya.txt
```

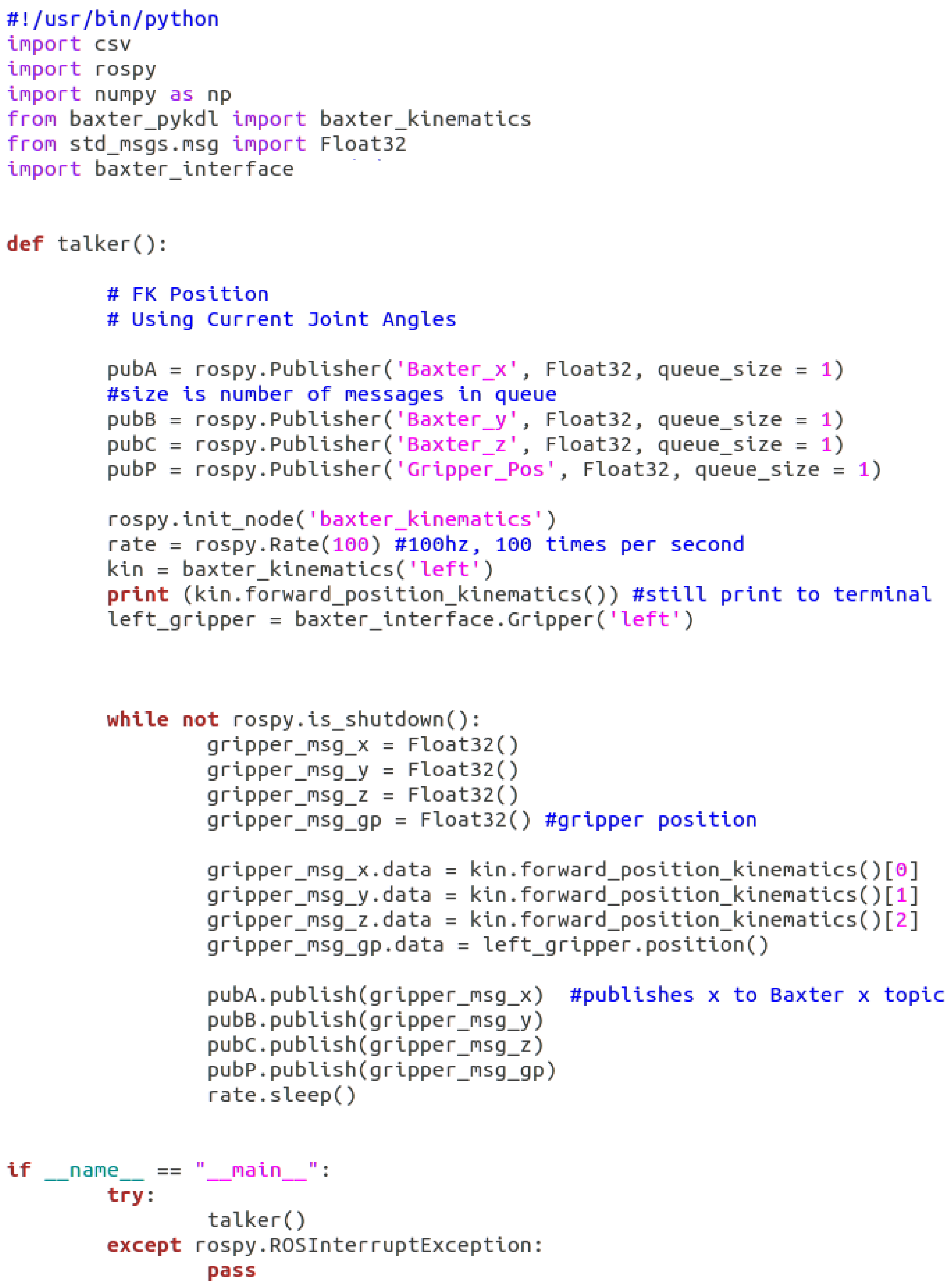
```
stephanie@solson-computer:~$ source ~
/catkin_workspace/devel/setup.sh
stephanie@solson-computer:~$ cd ~/cat
kin_workspace/
stephanie@solson-computer:~/catkin_wo
rkspace$ ./baxter.sh
[baxter - http://011303P0006.local:11
311] stephanie@solson-computer:~/catk
in_workspace$ rostopic echo chatterza
 | tee testza.txt
```

```
Blink | Arduino 1.8.3
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.
```

Fig. 25. Program windows and terminals

3.4 Obtaining and Communicating Gripper Coordinates

As mentioned in the literature review, the joint angles of Baxter’s arms could be obtained through the joint recorder or other means; however, forward kinematics were required to solve for the gripper’s location relative to Baxter’s base frame. Therefore, the Python code in Fig. 26 was written utilizing the forward position kinematics function in the Baxter PYKL package [31]. The function returns an array of seven numbers, the first three being the left gripper’s x, y, and z coordinates in meters. For the electric parallel gripper, the point returned is the fingertip location. The three coordinates were published separately to three different ROS topics. The degree to which Baxter’s gripper was open, with 0 being fully closed and 1 being fully open was published to a fourth topic.



```
#!/usr/bin/python
import csv
import rospy
import numpy as np
from baxter_pykdl import baxter_kinematics
from std_msgs.msg import Float32
import baxter_interface


def talker():

        # FK Position
        # Using Current Joint Angles

        pubA = rospy.Publisher('Baxter_x', Float32, queue_size = 1)
        #size is number of messages in queue
        pubB = rospy.Publisher('Baxter_y', Float32, queue_size = 1)
        pubC = rospy.Publisher('Baxter_z', Float32, queue_size = 1)
        pubP = rospy.Publisher('Gripper_Pos', Float32, queue_size = 1)

        rospy.init_node('baxter_kinematics')
        rate = rospy.Rate(100) #100hz, 100 times per second
        kin = baxter_kinematics('left')
        print (kin.forward_position_kinematics()) #still print to terminal
        left_gripper = baxter_interface.Gripper('left')



        while not rospy.is_shutdown():
                gripper_msg_x = Float32()
                gripper_msg_y = Float32()
                gripper_msg_z = Float32()
                gripper_msg_gp = Float32() #gripper position

                gripper_msg_x.data = kin.forward_position_kinematics()[0]
                gripper_msg_y.data = kin.forward_position_kinematics()[1]
                gripper_msg_z.data = kin.forward_position_kinematics()[2]
                gripper_msg_gp.data = left_gripper.position()

                pubA.publish(gripper_msg_x)  #publishes x to Baxter x topic
                pubB.publish(gripper_msg_y)
                pubC.publish(gripper_msg_z)
                pubP.publish(gripper_msg_gp)
                rate.sleep()


if __name__ == "__main__":
        try:
                talker()
        except rospy.ROSInterruptException:
                pass
```

Fig. 26. Python code to publish gripper information

3.3 Programming Baxter’s Movements

For testing, three trajectories were programmed for Baxter’s left gripper. For the first test, a circular trajectory was created parallel to the YZ plane just over one meter from Baxter’s base frame. After selecting the path’s center point and radius, eight points along the path were planned in Excel as shown in Fig. 27.

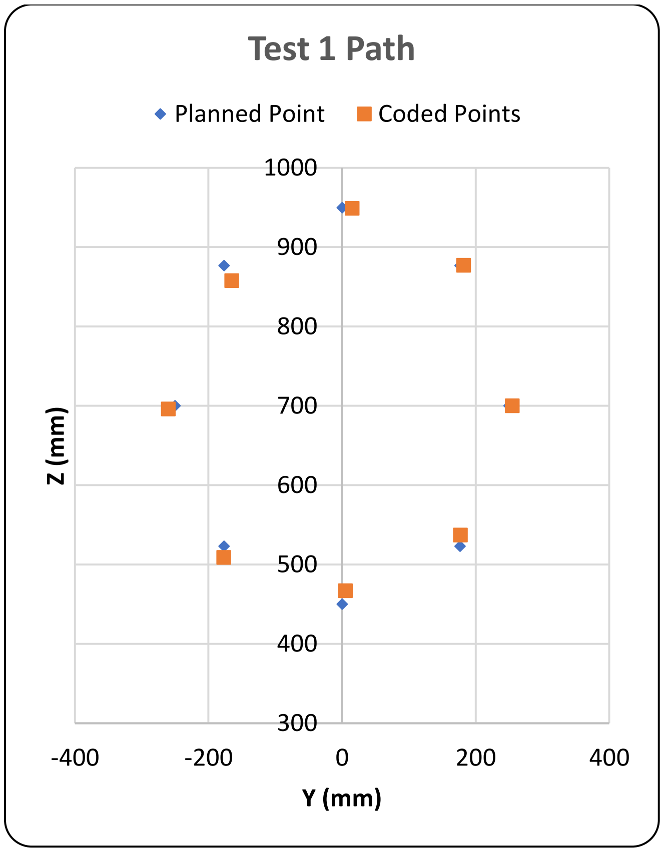


Fig. 27. Baxter Test 1: planned path and final coded path

Although Baxter’s PyKDL package has an inverse kinematics function to convert Cartesian points to joint angles for programming, attempts to use the function for gripper positions known to be within Baxter’s workspace returned no results. The process to program the trajectory, therefore, went as follows: Baxter’s left arm was first moved into the first desired pose by approximation and altered until the forward kinematics returned a Cartesian coordinate near the desired point. Current joint angles were obtained by echoing the ROS topic /robot/joint_states in a terminal window and copying them temporarily to a file. The process was repeated to take down the joint angles of all eight poses. After taking down all sets of joint angles, a simple Python file was written to move Baxter's gripper from one coded point to another. The code is in Appendix C. The path was simple: the gripper started at Point 1 and traversed the circular path twice clockwise and then twice counterclockwise. Pauses were coded to hold Baxter at each point for one second. For the gripper to move along the path, the Python file is run in the third terminal. The fourth terminal is not used.

The second trajectory was the object manipulation task shown in Fig. 28. In the task, Baxter reached for an object on a platform, picked it up, moved it to his right along an inverted parabolic path, set it down, released it, and raised his arm. The target position is where the object is set at the end of the task. To control the velocity of the task and prevent awkward pauses at instructed points, the trajectory needed to be programmed differently than the previous. Specifically, the joint recorder, joint trajectory action server, and joint trajectory file playback included in Baxter's software were utilized [32]. With the joint recorder running in a terminal, Baxter's left gripper was moved by hand at the desired speed while opening and closing the gripper when needed. The Python file wrote timestamped joint positions for both arms to a .csv file. For Baxter to repeat the trained movement, the joint trajectory action server is run in the third terminal followed by the joint trajectory file playback in the fourth terminal.

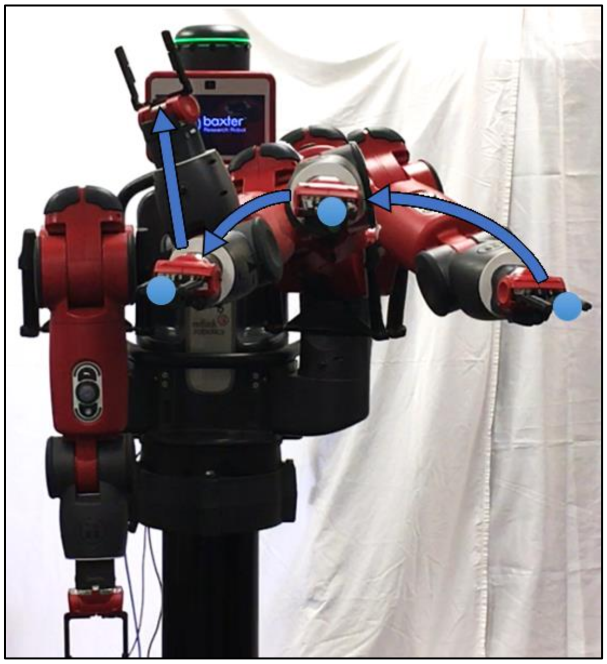


Fig. 28. Baxter inverted parabolic trajectory

Using the same programming method, Baxter was taught a third task shown in Fig. 29. Baxter reached for an elevated object, picked it up, moved down and to his left along a linear trajectory, set it down, released it, and raised his arm.

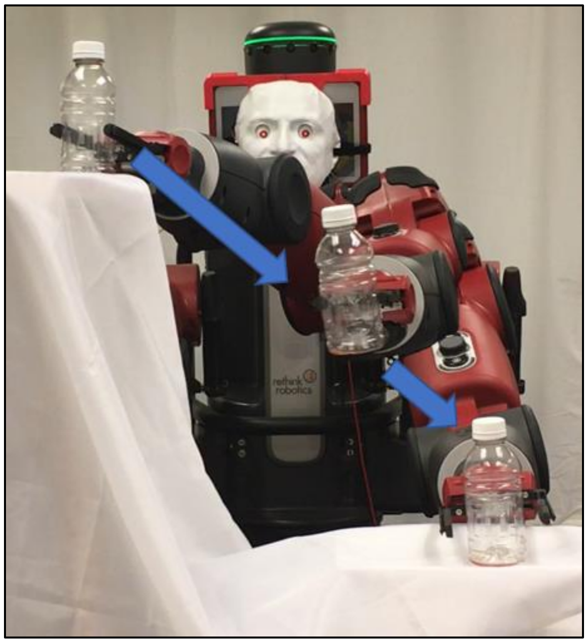


Fig. 29. Baxter linear trajectory

The grasp site, target, average velocity of the gripper in the y-direction, and the equation for z along a trajectory given y needed to be calculated for each trajectory and added to the Arduino sketch. Grasp site coordinates were found by taking the gripper's location when it closes above the grasp site and subtracting the measured vertical distance between the gripper and grasp site at the same moment. The targets were found in a similar fashion. Average velocities in y were calculated in Excel from the time-stamped data obtained through the joint recorder. Finally, the y- and z-values of each trajectory were plotted and curve fit, resulting in equations for z as a function of y. The coefficients of determination, R^2, associated with the parabolic and linear curve-fits were 0.96 and 0.98, respectfully.

3.4 Arduino

The Arduino code was organized into nine tabs. One header file and one CPP file were written for each of the following sections: DH parameters, servo and potentiometer, motor and encoder, and verification. The header files initialize variables and functions and place them into classes, and the CPP files contain the rest of the relevant code and reference the appropriate header files. The ninth tab is the Arduino sketch that calls the functions from the CPP files in void setup() and void loop() and defines what "mode" the eyes are operating under.

Apart from executing the functions in the CPP files, the sketch file also contains four subscribers. The subscribers listen to the ROS topics published to in the Python code and assign the x, y, and z coordinates of the left gripper and the degree to which the gripper is open to Arduino variables. The final Arduino code and pseudocode flowcharts are in Appendix D.

3.4.1 DH Parameters

The DH parameter code determines where the centroid of the bar in Fig. 30 needs to be for the eyes to look at a given point in space. The code then calculates the required displacement in x, y, and z from the centroid's current position to reach the desired position.

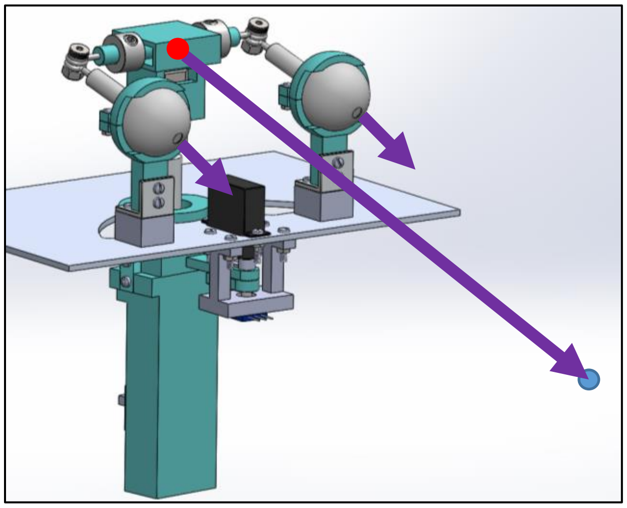


Fig. 30. Bar centroid (red) in relation to gaze fixation point (blue)

Four reference frames were used as shown in Fig. 31: Baxter's base reference frame, the midpoint between eyeball centroids, the gripper position, and the bar centroid. The Rethink Robotics website states that z is zero relative to his base frame "where the grey lower front panel meets the black metal that connects to the robot's pedestal" [33]. The default orientation of the base frame's x-, y-, and z-axes were applied to the other three frames. The notation P_b^a used in the coming calculations means the position of {b} relative to {a}.

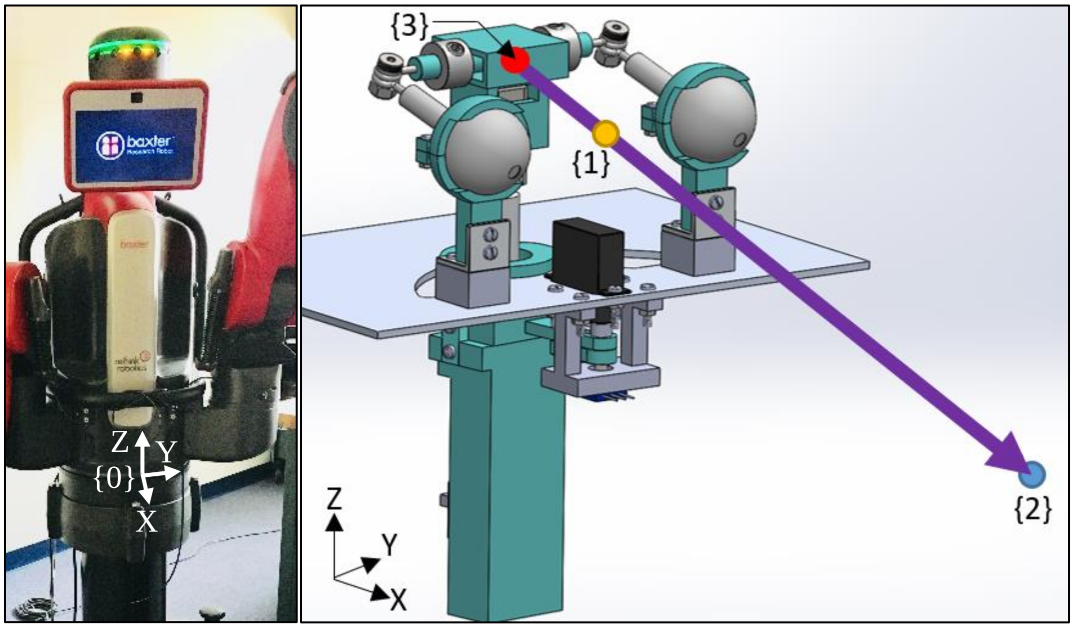


Fig. 31. Reference frames

First, the constant P_1^0 was chosen by measuring how high above Baxter's base frame the eyes would be mounted. The gaze fixation point P_2^0 was initialized as a point straight ahead of and level with the eyes. The position of the bar centroid relative the eyeball midpoint, P_3^1, was then initialized as the servo's moment arm length, 37mm, all in the negative x-direction. Note the magnitude of the vector between {1} and {3} is always 37mm. Calculations begin with finding P_2^1 using Equation 2.

$$P_2^1 = P_2^0 - P_1^0 = (x_2 - x_1)\hat{\imath} + (y_2 - y_1)\hat{\jmath} + (z_2 - z_1)\hat{k} \tag{2}$$

The magnitude of the vector between the bar centroid and gripper position, $\|\vec{A}\|$, is found using Equation 3.

$$\|\vec{A}\| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{3}$$

Equations 4 and 5 are equivalent definitions of the unit vector in the direction of $\vec{A}$, where α, β, and γ are the angles from the x-, y-, and z-axes, respectfully, to $\vec{A}$.

$$\vec{u} = \cos(\alpha)\,\hat{\imath} + \cos(\beta)\,\hat{\jmath} + \cos(\gamma)\,\hat{k} \tag{4}$$

$$\vec{u} = \frac{x_2^1}{\|\vec{A}\|}\hat{\imath} + \frac{y_2^1}{\|\vec{A}\|}\hat{\jmath} + \frac{z_2^1}{\|\vec{A}\|}\hat{k} \tag{5}$$

The components of Equations 4 and 5 were equated and rewritten to solve for α, β, and γ as shown in Equation 6.

$$\alpha = cos^{-1}\left(\frac{x_2 - x_1}{\|\vec{A}\|}\right)$$

$$\beta = cos^{-1}\left(\frac{y_2 - y_1}{\|\vec{A}\|}\right)$$

$$\gamma = cos^{-1}\left(\frac{z_2 - z_1}{\|\vec{A}\|}\right) \tag{6}$$

The three angles give the orientation of the vector that passes through {1}, {2}, and {3}. Knowing Equation 6 can be written for any pair of {1}, {2} and {3} coordinates resulted in Equation 7. x_3, y_3, and z_3 are the components of P_3^0, and r = 37mm.

$$x_3 = x_1 - rcos(\alpha)$$

$$y_3 = y_1 - rcos(\beta)$$

$$z_3 = z_1 - rcos(\gamma) \tag{7}$$

To condense the code, the equations for α, β, and γ were plugged into Equation 7 to form Equation 8 used in the final code.

$$x_3 = x_1 - r\left(\frac{x_2 - x_1}{\|\vec{A}\|}\right)$$

$$y_3 = y_1 - r\left(\frac{y_2 - y_1}{\|\vec{A}\|}\right)$$

$$z_3 = z_1 - r\left(\frac{z_2 - z_1}{\|\vec{A}\|}\right) \quad (8)$$

The new value of P_3^1 is found using Equation 9.

$$P_3^1 = P_3^0 - P_1^0 = (x_3 - x_1)\hat{\imath} + (y_3 - y_1)\hat{\jmath} + (z_3 - z_1)\hat{k} \quad (9)$$

Equation 10 gives Δx, Δy, and Δz for the bar centroid.

$$\Delta x = x_3^1 - x_{3\,old}^1$$

$$\Delta y = y_3^1 - y_{3\,old}^1$$

$$\Delta z = z_3^1 - z_{3\,old}^1 \quad (10)$$

It is important to guarantee that all gaze fixation points are within the eyes' range of motion. The eyes vertical range of motion is shown in Fig. 32 in which the upper and lower limit occurs when a shaft coupler encounters either the press-in nut or the horizontal surface of the encoder track.

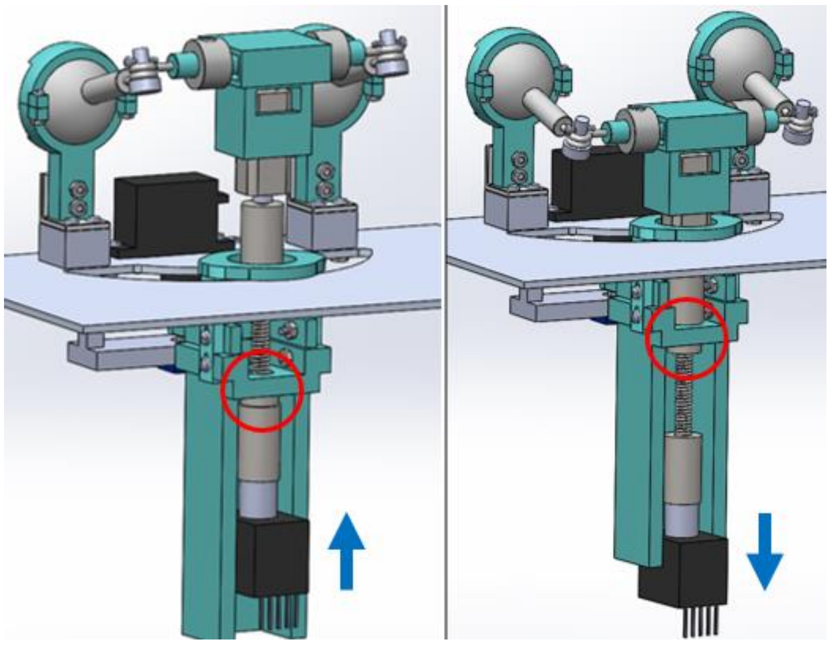


Fig. 32. Limitations to vertical eye movement

To avoid collisions, all gaze fixation points were chosen for L shown in Fig. 33 to remain within 11mm. The conservative limitation reduces the eyes' vertical range of motion from 40° to 35°.

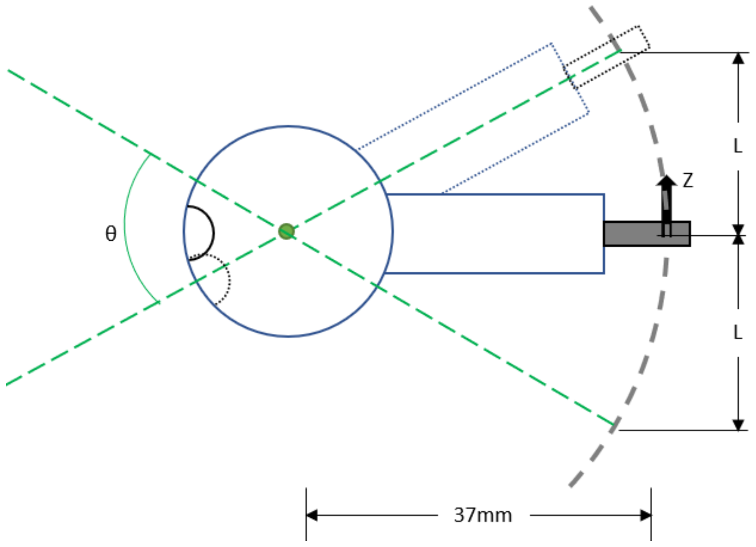


Fig. 33. Eyeball rotation about its centroid in the xz-plane

The eye's vertical range of motion, θ, is related to L through Equation 11.

$$\theta = 2sin^{-1}\left(\frac{L}{37mm}\right) \quad (11)$$

3.4.2 Servo – Potentiometer

Fig. 34 shows the parameters used to calculate the angle, Δθ, between the initial and final position of the bar centroid. Because the servo used was not programmable, the servo arm was attached so its angle would be as close to 90° as possible when the eyes were aimed straight ahead. Once in the assembly, angle commands were sent to the servo by trial and error to close in on the actual angle needed to aim the eyes forward. The result was 97°.

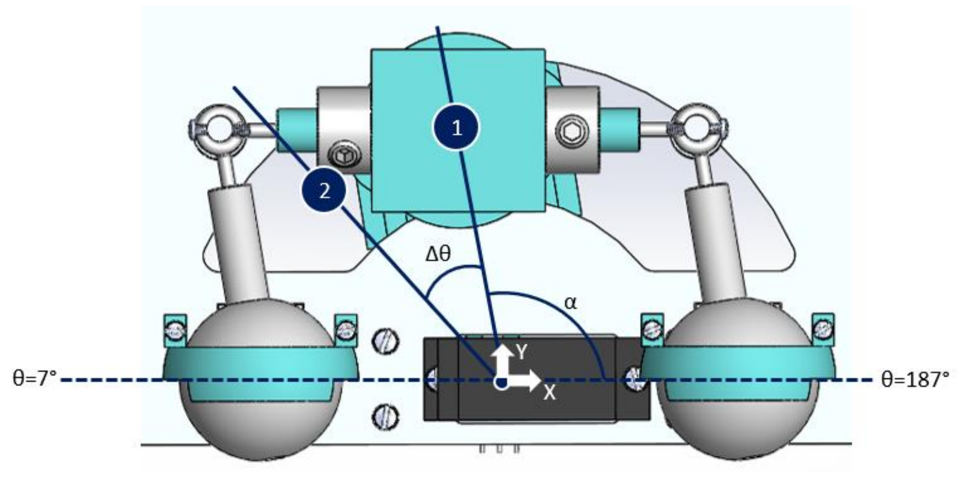


Fig. 34. Parameters used to calculate Δθ

Equations 12 through 15 were used to find θ_2 in degrees. The values of y_3^1 and z_3^1 were calculated in the DH parameter section of code. The eyes' radius of rotation, or the distance from {1} to {3} is r = 37mm, and r_{xy} is the length of the radius projected onto the XY-plane. Note that in the final code, Equations 12 through 14 were combined into one equation to calculate Δθ.

$$\alpha = (187° - \theta_1) \tag{12}$$

$$r_{xy} = \sqrt{r^2 - z_3^1} \tag{13}$$

$$\Delta\theta = cos^{-1}\left(\frac{y_3^1}{r_{xy}}\right) - \alpha \tag{14}$$

$$\theta_2 = \theta_1 + \Delta\theta \tag{15}$$

The limits to the servo's range of motion are shown in Fig. 35. The analog read code in Appendix E was used to obtain the potentiometer's output for each limit.

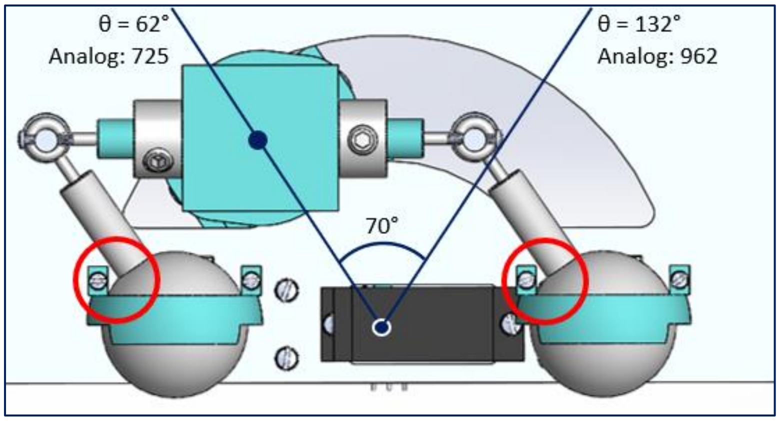


Fig. 35. Parameters to map potentiometer analog output to degrees

Using the angles and corresponding potentiometer outputs of the limits, Equation 16 can interpolate any value of θ_1 in degrees given the potentiometer output. The equation was used in place of an Arduino map() function since the latter uses integer math that would have led to roundoff errors [34].

$$\theta_1 = 132° - \frac{(962 - analog\ output)(132° - 62°)}{962 - 725} \qquad (16)$$

The Servo library built into Arduino prevents pins 9 and 10 from acting as PWM pins [35]. Because the motor shield uses those pins as PWM pins, and because those motor shield pins cannot be remapped, the open source ServoTimer2 Arduino library was used in place of the Servo library. Unlike the built-in library, the ServoTimer2 library controls a servo by writing an input in milliseconds instead of degrees to the servo. Although the range 1000-2000ms can roughly be mapped to 0-180°, the true mapping varies between servos. Therefore, the two numbers 1300 and 1800 milliseconds were chosen, and the analog read code was used to get the equivalent servo degrees for each. The results were 80° and 128°, respectfully. The four numbers were used in Equation 17 to make a second interpolation equation.

$$\theta_1[ms] = 1800 - \frac{(128° - \theta_1)(1800 - 1300)}{128° - 80°} \tag{17}$$

When an angle is written to a servo, no other code can run until the command has finished executing. For this reason, the servo instructions were sent in increments of one degree. Because the Arduino delay() function also hinders any other code from running until its completion, the millis() function was used to track time. The amount of time allotted between servo increments, t_i, was found using Equation 18 where the constant T is the total time allotted for the servo to move from its initial to final position. An if statement based on t_i was used to control when a servo angle was incremented.

$$t_i = \frac{\Delta\theta}{T} \tag{18}$$

The logic for the servo is follows: first, θ_1, $\Delta\theta$ and the length of time between each servo command is calculated. The time between commands is set to a minimum of two milliseconds so the servo is not instructed to move at a speed faster than its specified maximum. If the calculated time for a servo command is reached, the calculation for θ_2 is run. θ_2 is then constrained within the servo's range of motion. The servo's current position is calculated by taking the potentiometer output and converting it to an angle in degrees. If the servo's current position is over one degree from the desired θ_2, the servo is told to move one degree closer to θ_2. Otherwise, θ_2 is written to the servo. θ_2 is also written to the servo if the timer goes 50ms past the allotted 100ms for servo movement. The process repeats with every new gaze fixation point instruction.

3.4.3 DC Motor – Encoder

The encoder code was open-source from Arduino [36]. It attaches interrupts to the pins associated with the encoder's Channel A and B, respectively. The encoder returns quadrature output; instead of having a continuous reading, a number is returned based on the number of encoder “ticks” that occur when the shaft rotates. The encoder’s specifications state it has a resolution of 100ticks/rev; however, because the rise and fall of both channels is being tracked in the code, the resolution increased to 400ticks/rev [36]. The output can be positive or negative depending on whether the encoder’s net rotation is clockwise or counterclockwise from its position when the Arduino sketch was first uploaded. In the assembly, a positive Δz decreases the number returned while a negative Δz increases it.

An Arduino library and example sketch were downloaded from GitHub for the Pololu motor driver shield [37]. One of the library’s built-in functions used stops the motor if a fault is found. An example of a fault would be if the voltage reaching the motor was less than the operating voltage.

Another function from the same library sets the DC motor’s speed and direction of rotation. Instead of inputting a speed in RPMs, the motor speed is set using a speed index between -400 and 400 where the sign differentiated clockwise from counterclockwise. Therefore, a function relating the speed index to RPMs had to be determined empirically as explained in Section 5.3.2.

The motor’s loop function first calculates the number of motor rotations, N, required to cover the instructed Δz using Equation 19.

$$N = \Delta z * 24 \frac{rot}{inch} * \frac{1inch}{25.4mm} \tag{19}$$

Δz is in millimeters and comes from the DH parameter calculations, 24 rotations-per-inch comes from the threads-per-inch of the screw, and the final term is a unit conversion. The loop goes on to calculate the time allotted for acceleration as a set portion of the predefined time given for the motor to move, the motor’s maximum RPM constrained within the motor’s specifications, and the RPM increment. A new variable equivalent to the encoder output is defined for the verification code to reference.

The motor’s speed at a given time is calculated to fit an asymmetrical trapezoidal velocity profile. The profile was chosen through testing explained in Section 5.3.3. Besides conditions for acceleration and deceleration, a condition that gave feed forward control was included. The condition recalculates the desired maximum RPMs at the end of the acceleration phase if the motor is moving slower than anticipated. When none of the three conditions are accessed, the motor velocity remains constant, making up the plateau of the velocity profile. A correction function is accessed after the allotted time of 115ms for the motor reach its destination has passed. The function runs the motor at a speed index of ±125, or 67RPM, in the appropriate direction to make up for any error in position. A separate function stops the motor if the vertical motor’s position is within the allowed error of 0.1mm. The correction function is ended when the timer reaches 300ms.

Like the servo angle, the DC motor speed was changed in increments of time. The servo’s time increment, however, was calculated based on the total Δθ while the servo angle increment was constant at one degree regardless of the time allotted for motor movement. In contrast, the DC motor’s time increment was a constant while the motor speed’s increment was calculated to fit the velocity profile.

3.4.4 Verification

The verification section of code does what its name suggests; it verifies the orientation of the assembly at a given time. It uses the servo and encoder outputs to calculate two points: the location of the bar centroid and where the eyes are looking.

When the code starts, the eyes are looking straight ahead, meaning the bar centroid's z relative to the eye midpoint is zero, or $z_3^1 = 0$. The actual value, $z_{3,a}^1$, can be calculated anytime during a run by converting encoder ticks to millimeters using Equation 20. A negative sign is required because a positive z_3^1 corresponds to a negative encoder output.

$$z_{3,a}^1 = -eTicks * \frac{1\,rev}{400\,ticks} * \frac{25.4mm}{24\,rev} \tag{20}$$

After converting the potentiometers analog output to degrees to get θ_2, y_3^1 is calculated in Equation 21 where r = 37mm.

$$y_{3,a}^1 = \sqrt{r^2 - \left(z_{3,a}^1\right)^2} * \cos(187° - \theta_2) \tag{21}$$

Having r, $y_{3,a}^1$ and $z_{3,a}^1$, $x_{3,a}^1$ was found using Equation 22. $x_{3,a}^1$ is negative because the bar centroid is always behind the eye midpoint.

$$x_{3,a}^1 = -\sqrt{r^2 - \left(y_{3,a}^1\right)^2 - \left(z_{3,a}^1\right)^2} \tag{22}$$

It's odd to say the exact point that the eyes are "looking" at can be found; a single eye orientation has a vector normal to the pupil along which are infinite possible fixation points. To reduce the vector to one point, the actual x-value of the fixation point was equated to that of the instructed point. Therefore, the two points lie in a plane normal to the x-direction as shown in Fig. 36, making it easy to compare them.

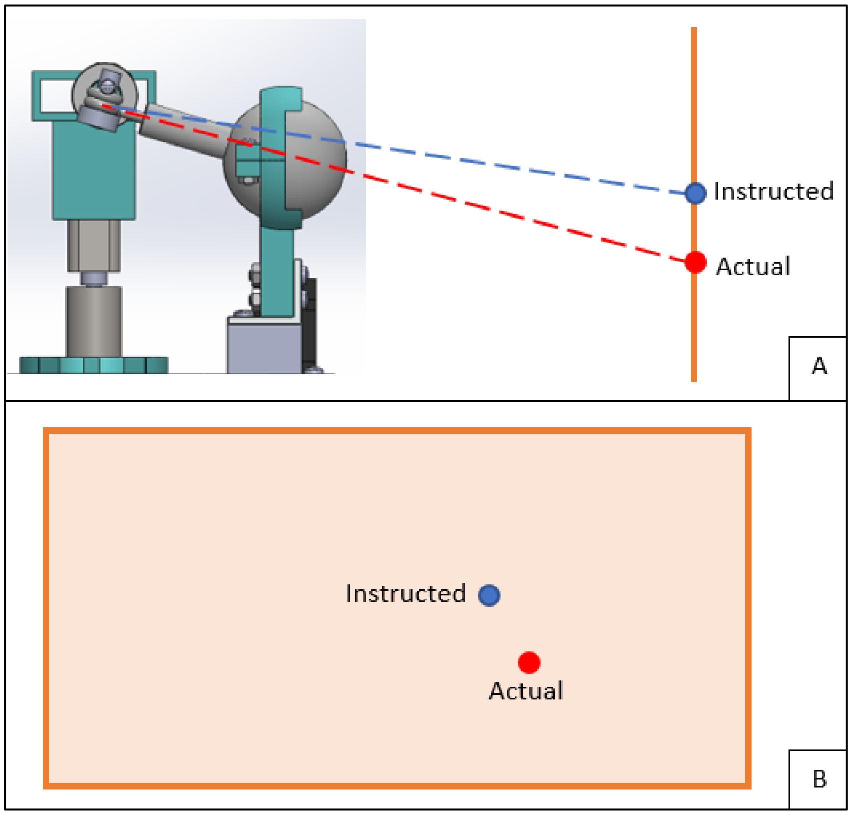


Fig. 36. Instructed versus actual gaze fixation point in yz-plane

For the full calculation of $x_{2,a}^{0}$, $y_{2,a}^{0}$, and $z_{2,a}^{0}$, the actual fixation point's x-values relative to {0} and {1} were equated to their respective input points in Equations 23 and 24.

$$x_{2,a}^{0} = x_{2}^{0} \tag{23}$$

$$x_{2,a}^{1} = x_{2}^{1} \tag{24}$$

Next, the actual location of the bar centroid was calculated relative to the base frame in Equation 25.

$$\begin{gathered} x_{3,a}^{0} = x_{3,a}^{1} + x_{1}^{0} \\ y_{3,a}^{0} = y_{3,a}^{1} + y_{1}^{0} \\ z_{3,a}^{0} = z_{3,a}^{1} + z_{1}^{0} \end{gathered} \tag{25}$$

The cosines of the angles that define the vector $\vec{A}_a$ from the fixed eye midpoint to the actual bar centroid were calculated in Equation 26, and the magnitude of the vector was calculated in Equation 27.

$$\cos(\alpha) = \frac{x_1^0 - x_{3,a}^0}{r}$$

$$\cos(\beta) = \frac{y_1^0 - y_{3,a}^0}{r}$$

$$\cos(\gamma) = \frac{z_1^0 - z_{3,a}^0}{r} \qquad (26)$$

$$\left\|\vec{A}_a\right\| = \frac{x_{2,a}^1}{\cos(\alpha)} \qquad (27)$$

Finally, after calculating $y_{2,a}^1$ and $z_{2,a}^1$ in Equation 28, $y_{2,a}^0$ and $z_{2,a}^0$were found using Equations 29.

$$y_{2,a}^1 = \left\|\vec{A}_a\right\| * \cos(\beta)$$

$$z_{2,a}^1 = \left\|\vec{A}_a\right\| * \cos(\gamma) \qquad (28)$$

$$y_{2,a}^0 = y_{2,a}^1 + y_1^0$$

$$z_{2,a}^0 = z_{2,a}^1 + z_1^0 \qquad (29)$$

3.4.5 Modes

The Arduino code was written for three modes of eye movement. Mode 1 equates the left gripper position to the instructed gaze fixation point; i.e. the eyes always look at the gripper. Mode 1 can be used for any gripper position within the eyes range of motion.

The remaining two modes give the eyes more human-like eye movement but are only programmed to work for the object manipulation tasks discussed in Section 3.3. The modes are called block mode and full glass mode. As the mode names suggest, Baxter's eye movements in full glass mode make him appear to be handling the object more cautiously than in block mode.

In both modes, the eyes are programmed to look forward until the Python code begins publishing left gripper information to ROS topics. Then, the eyes look at the grasp site as the gripper moves towards the object. After the gripper is thirty-percent closed just before grabbing the object, the eyes switch between fixating on the object's desired final position, referred to as the target, T, and a temporally offset point ahead of the gripper. The first target fixation always occurs right after the gripper is thirty-percent closed, and subsequent fixations occur at fixed time intervals. Both aspects keep eye movement consistent between trials. A special time condition was added to the Arduino code when the linear trajectory was run to ensure the eyes were fixating the target and not an offset point before the gripper opened. The condition was not necessary in the parabolic trajectory due to the timing of the saccades. In the middle of the gripper releasing the object, the eyes looked straight ahead to indicate the completion of the task.

The two differences between the modes are as follows: the target is fixated more frequently and thus for a greater portion of the total time in full glass mode, and the temporal offset in full glass mode is less than that of block mode. The temporal offsets chosen for block mode and full glass mode were 1.5s and one second, respectively. The equivalent distance offsets are calculated by multiplying the temporal offsets by the known average gripper velocity in the y-direction. Note the parabolic trajectory was stretched far enough in the y-direction that the velocity in z was negligible. When the conditions are met to look at a point offset from the gripper, the offset y-value is calculated and plugged into the equation to find the offset value of z.

3.4.6 Output

Eight publishers were coded in the Arduino sketch file, one for each of the following: the time in milliseconds, the instructed gaze fixation point coordinates X, Y, and Z, the actual gaze fixation point coordinates Y_{EYES} and Z_{EYES}, and the left gripper's coordinates Y_{HAND} and Z_{HAND}. Note that $X_{HAND} = X_{EYES} = X$, which is why there is only a publisher for X. All coordinates are returned relative to the base frame; however, the code has the calculations done if the user wanted to return the instructed and actual gaze fixation points relative to the point between the eyeballs. To store the data, six of the eight topics can be echoed in their own terminals and written to texts files by executing a single command in each terminal. For reasons yet to be determined, running all eight publishers in the Arduino code created communication issues over the serial port. The way in which data was recorded and analyzed to work around the issue is discussed later.

3.4.7 Summary

Fig. 37 summarizes the Arduino inputs and logic. The mode predefined by the user and the gripper's location and percent open are used to determine the desired gaze fixation point. The algorithm goes on to calculate the horizontal and vertical bar displacements required to look at that point given the eyes' current gaze fixation point. The displacements dictate the movements of the lateral and vertical motors. After the time given for the eyes to reach their desired orientation, forward kinematics are performed to return the updated current gaze fixation point for the next run of the algorithm.

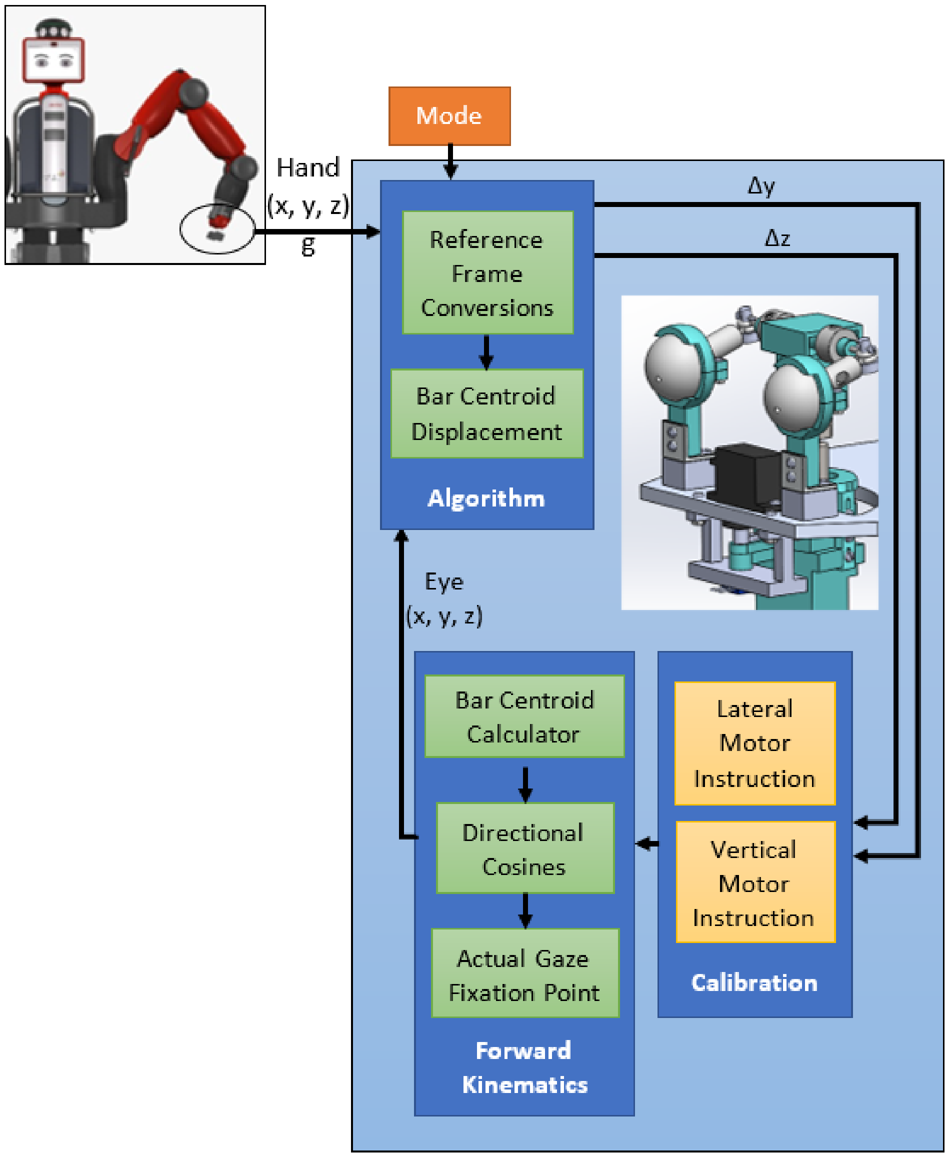

Mode
Hand
(x, y, z)
g
Δy
Δz
Reference Frame Conversions
Bar Centroid Displacement
Algorithm
Eye
(x, y, z)
Bar Centroid Calculator
Directional Cosines
Actual Gaze Fixation Point
Forward Kinematics
Lateral Motor Instruction
Vertical Motor Instruction
Calibration

Fig. 37. Arduino inputs and logic

4 FABRICATION

4.1 Preliminary

Excluding the mask and eyes for housing laser diode modules, Fig. 38 specifies all parts and quantities that were 3D printed for the assembly. The SolidWorks part files were converted to STL files, and the parts were printed on an Ultimaker 3 using Cura software. All pieces were printed using PLA and a twenty-percent infill.

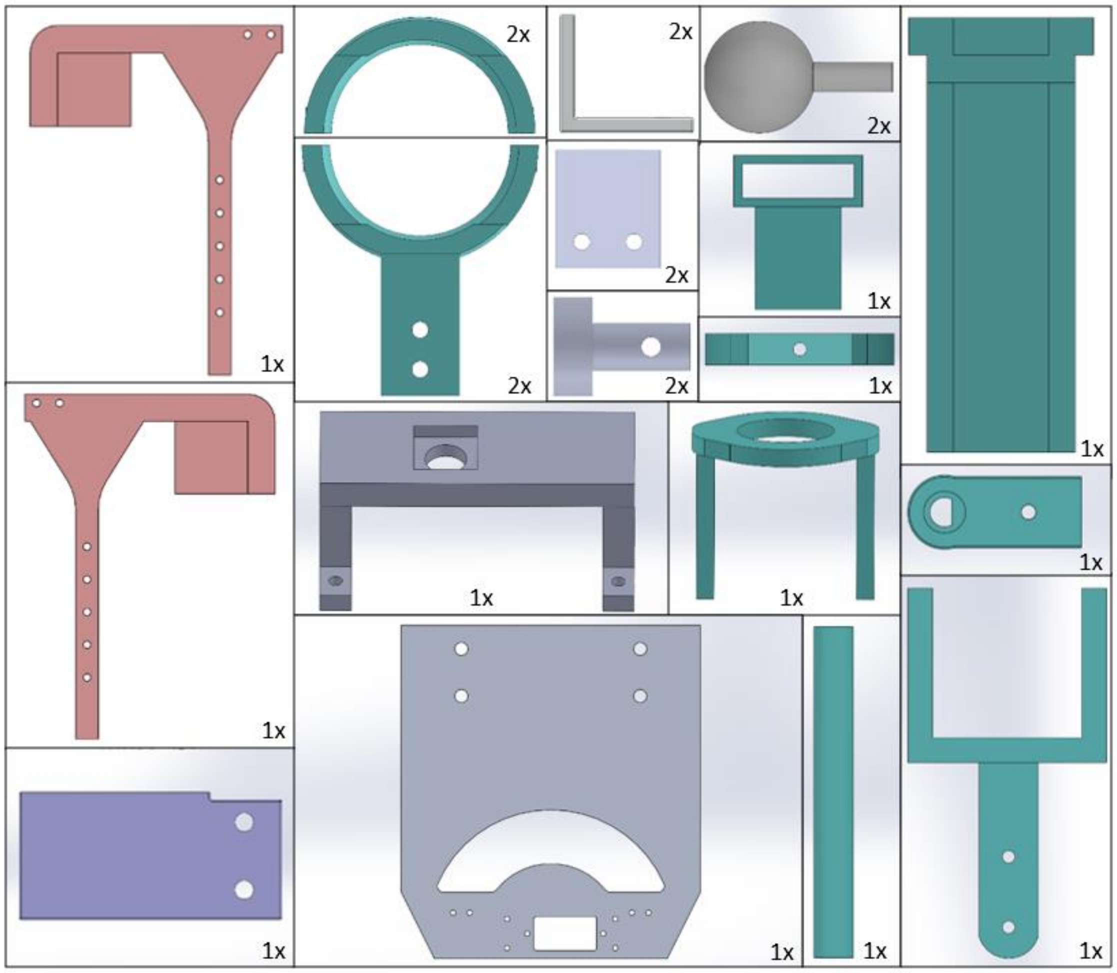


Fig. 38. 3D printed parts and quantities of each

After printing, many of the holes needed to be enlarged to meet the modeled dimensions. The holes were enlarged by hand using the drill bits shown in Fig. 39. Some parts like the plate were sanded to reduce friction. A file was used to smooth out areas that were printed with PLA supports like the DC motor housing slot.

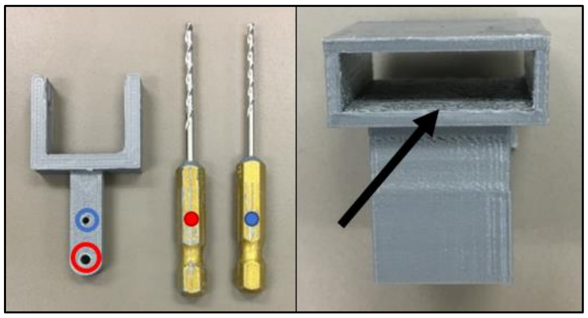


Fig. 39. Drill bits for enlarging holes (left), filed slot of DC motor housing (right)

The 10-32 threaded rod was made by sawing the head off a 2-1/2” screw as shown in Fig. 40. The final length was two inches, and the cut end was smoothed out on a sander.

Fig. 40. Original versus cut and sanded screw

Two parts needed soldering: the DC motor and the motor shield shown in Fig. 41. The DC motor had soldering tabs for its two terminals. The wires were initially sized based on the motor’s stall current; however, the 21AWG wire was too rigid to move within the assembly, which caused the soldering tabs on a test motor to snap. 26AWG wire was chosen knowing that if the motor ever stalled, the user would quickly stop the code, causing no damage to the thin wires. Next, three power blocks and four of the five included female headers were soldered to the motor shield.

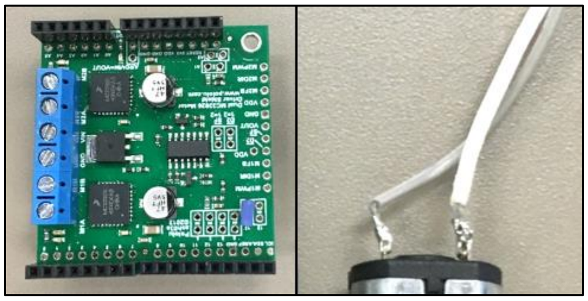


Fig. 41. Motor driver with soldered headers and power blocks (left); wires soldered to DC motor (right)

4.2 Assembly: Robotic Eyes

The beginning of the assembly in Fig. 42 focuses on the eyes. An eyebolt with an inner diameter of 1/8" was screwed into the cylindrical extrusion of an eye. The two-piece shell was then placed around the eyeball and secured with two M1.6 screws and nuts. The process was repeated for the other eye.

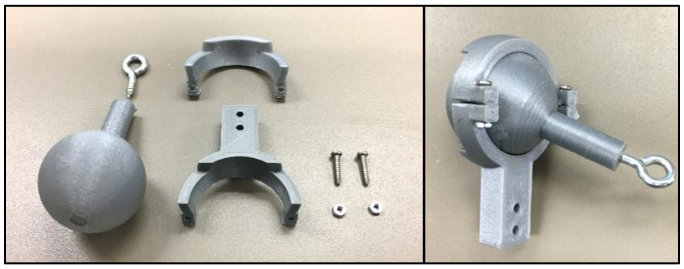


Fig. 42. Eyeball and housing disassembled (left) and assembled (right)

The L-brackets and standoffs shown in Fig. 43 were mounted to the plate with M2 x 20mm screws and nuts. Note that a thinner, rectangular version of the plate is depicted; however, the assembly is the same using the final plate. The eyeball assemblies from the previous step were then mounted to the L-brackets using M2 x 8mm screws and nuts.

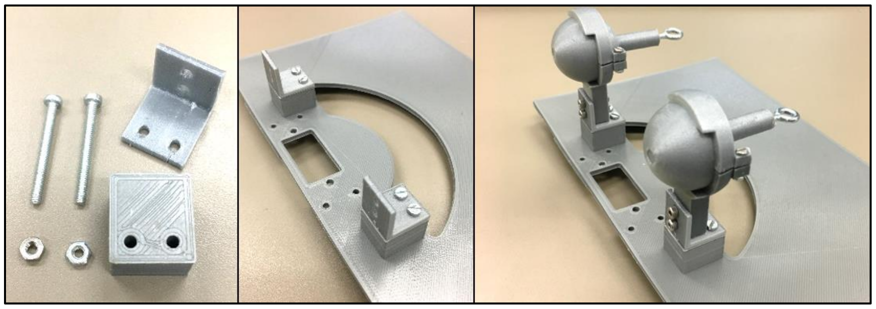


Fig. 43. L-brackets and standoffs used to mount the eyeball subassemblies

After positioning the servo horn on the servo as discussed in Section 3.4.2, the servo was mounted to the plate using two M2 x 8mm screws as shown in Fig. 44. To align the servo arm on top of the horn, an M1.6 screw was placed through the arm and horn while the screw for the servo shaft was inserted and tightened. The M1.6 screw was then removed and set aside for later.

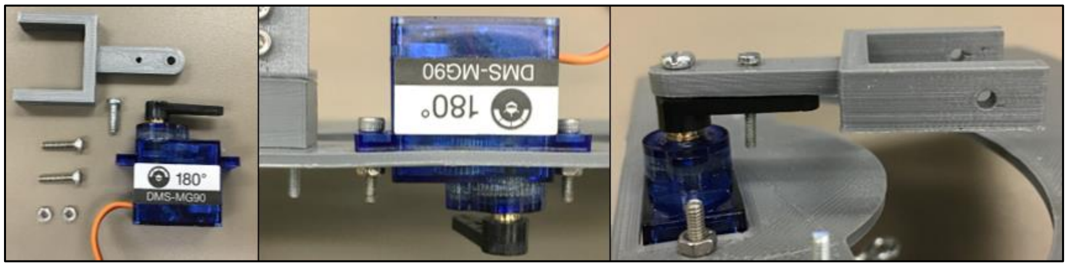


Fig. 44. Servo mounted to the plate followed by the servo arm

The potentiometer was press-fit into the square cutout of its mount as shown in Fig. 45. The potentiometer and servo interface in the bottom left of the figure were placed on the servo arm with the round cutout going over the servo shaft screw head. The M1.6 screw from the last step and a nut were used to connect the interface, servo arm and horn. The potentiometer assembly was then mounted to the bottom of the plate with four M2 x 8mm screws once the potentiometer shaft was fit through the semicircular hole in the interface piece.

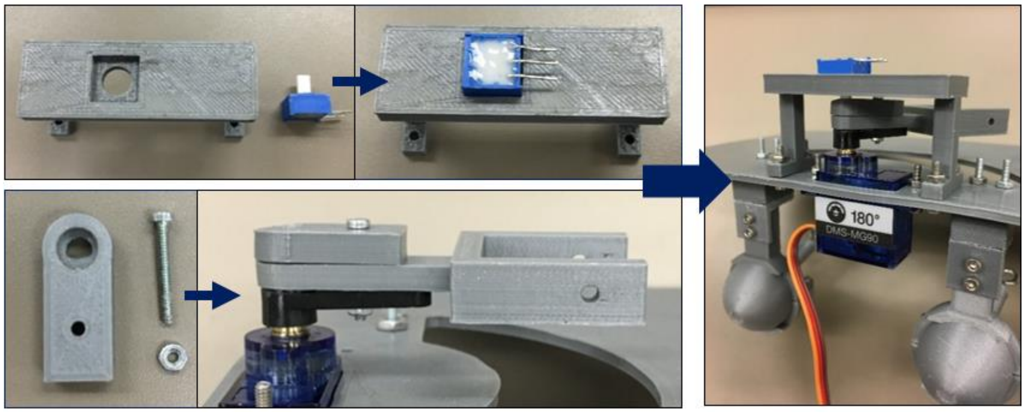


Fig. 45. Attaching the potentiometer and its mount and the servo-pot interface

Fig. 46 shows the hollow disk, support piece, and encoder track that were added next. Each of the three pieces were secured using two M2 x 8mm screws; the hollow disk was screwed to the servo arm, and the support piece and encoder track were screwed between the hollow disk extrusions.

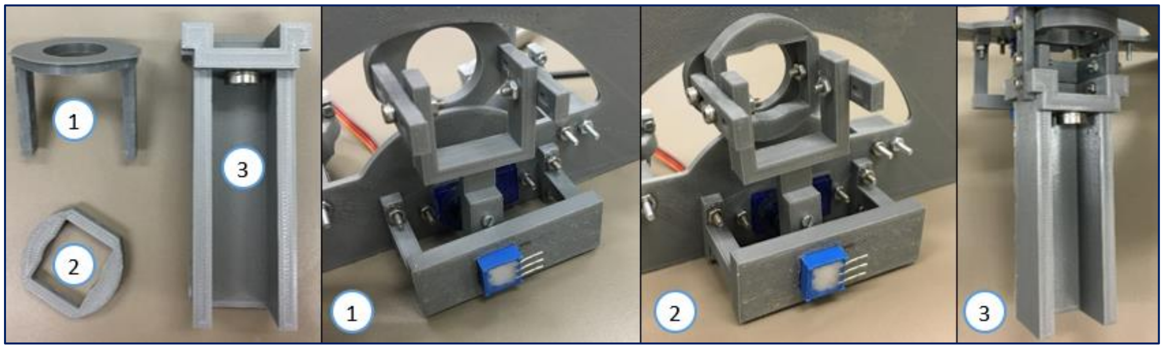


Fig. 46. Attaching the hollow disk (1), support piece (2), and encoder track (3)

Next, the shaft coupler for the DC motor and threaded rod was secured to one end of the rod using an Allen wrench as shown in Fig. 47. The free end of the rod was placed through the arced cutout in the plate and threaded through the press-in nut. The remaining coupler was then secured to the encoder shaft and then attached to the free end of the rod.

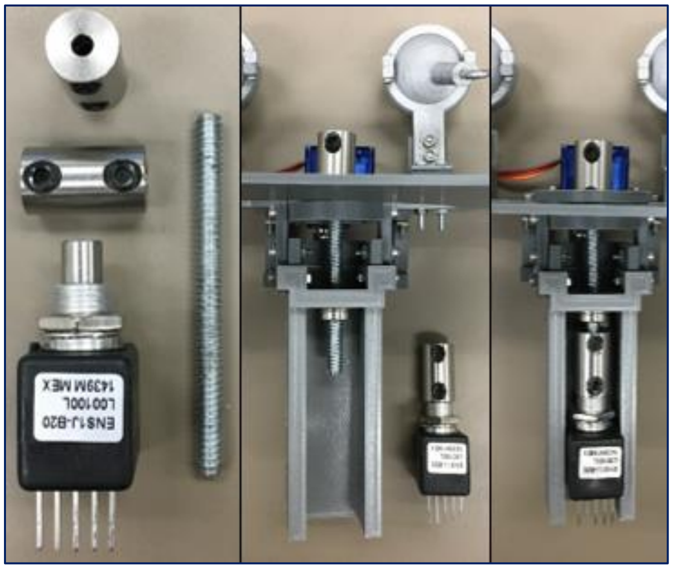


Fig. 47. Attaching the threaded rod, shaft couplers, and encoder

An eyebolt was screwed into one end of the bar shown in Fig. 48, and one shaft collar was slid onto the bar. The free end of the bar was slid into the slot on top of the DC motor housing. The second shaft collar was then slid onto the bar, and the second eyebolt was screwed into the free end of the bar. The motor housing was then centered along the bar, and the shaft collars were secured to the bar using an Allen wrench. Enough clearance was left between the housing and shaft collars for the bar to slide smoothly within the slot but not so much that the bar could twist about its centroid's z-axis. The spacing was controlled by putting paper between the housing and final shaft collar during assembly. The DC motor wires were then threaded into the bottom of the housing and out through one of the side openings. The motor was press-fit into the housing while ensuring the motor's soldering tabs did not hit the end of the cavity.

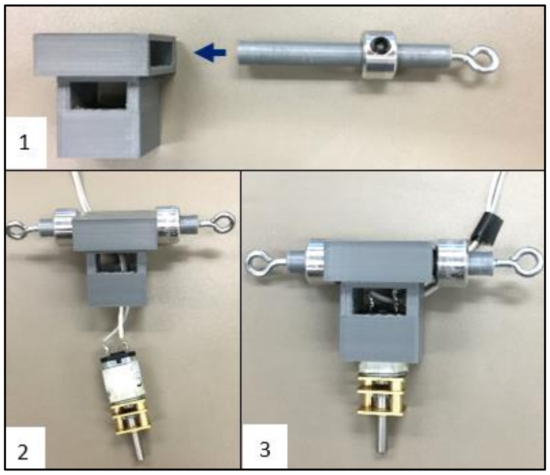


Fig. 48. Subassembly of the bar, eyebolts, shaft collars, DC motor and housing

The DC motor shaft was placed inside of the coupler and secured as shown in Fig. 49. With the eyeball eyebolts sitting on top of the bar eyebolts, the pins were inserted through the bolts, and an M1.6 screw was used to compete each pin connection.

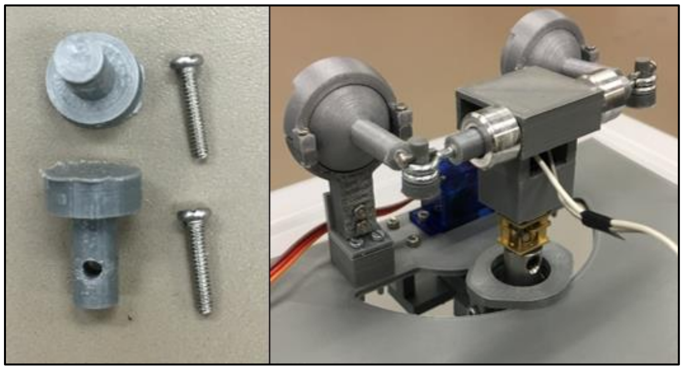


Fig. 49. Clevis pin connections

Without a lubricant on the threaded rod, the vertical motor would occasionally stall. To fix this, a few milliliters of WD-40 were sprayed through a nozzle into a small container shown in Fig. 50. Using a Q-tip, the lubricant was applied to the exposed portion of the rod between the two couplers.

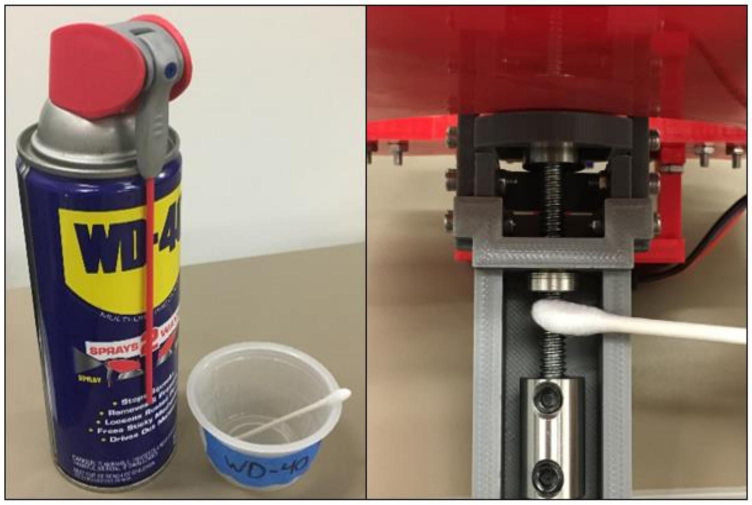


Fig. 50. WD-40 application to threaded rod

To construct each laser diode eye, the two-piece eye was 3D printed using white PLA. A slit was cut into an adhesive hole reinforcement as shown in Fig. 51. The reinforcement was then colored with a marker to look like a human iris. The reinforcement was centered on the front of the eyeball. One cut end of the reinforcement slightly overlapped the other for the reinforcement to attach flushed to the eyeball's curved surface. The area of the eye within the inner diameter of the reinforcement was colored to match the reinforcement. A pupil of an approximate diameter of 3.5mm was dotted on using a black permanent marker.

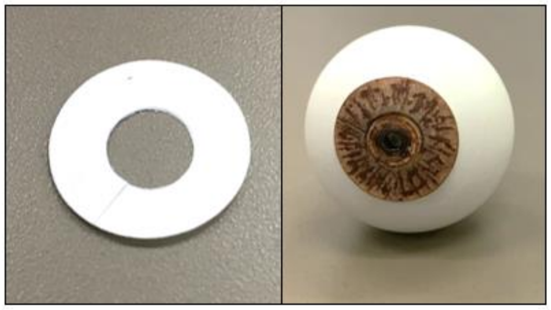


Fig. 51. Creating a laser diode eye's iris and pupil

As with the other eyes, an eyebolt was screwed into the cylindrical extrusion as shown in Fig. 52. The laser diode terminal wires were shortened, leads were soldered onto them, and heat shrink was put on the newly soldered connections. The laser diode was then placed in the eyeball. The leads were thread through the elliptical hole in the side of the cylindrical extrusion, and the two pieces of the eye were snapped together.

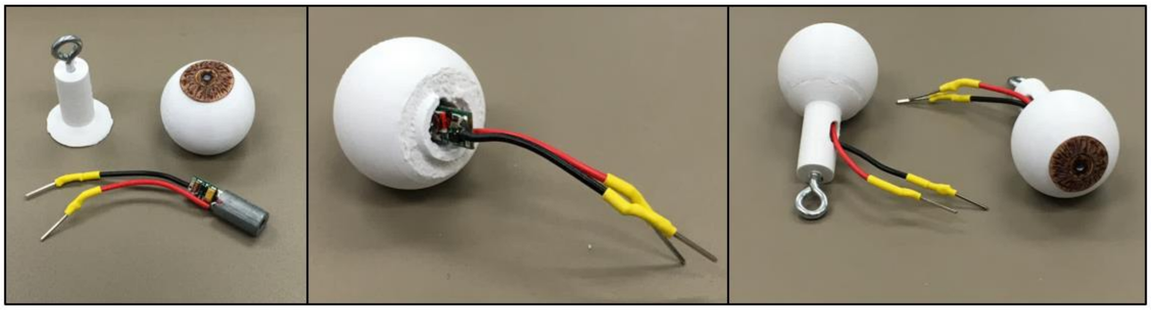


Fig. 52. Assembling laser diode eyes

4.3 Assembly: Wiring and Mounting

Fig. 53 shows most of the parts required for mounting the breadboard and mounting the plate to Baxter's display screen. Four #8-32 screws and nuts not pictured were also used.

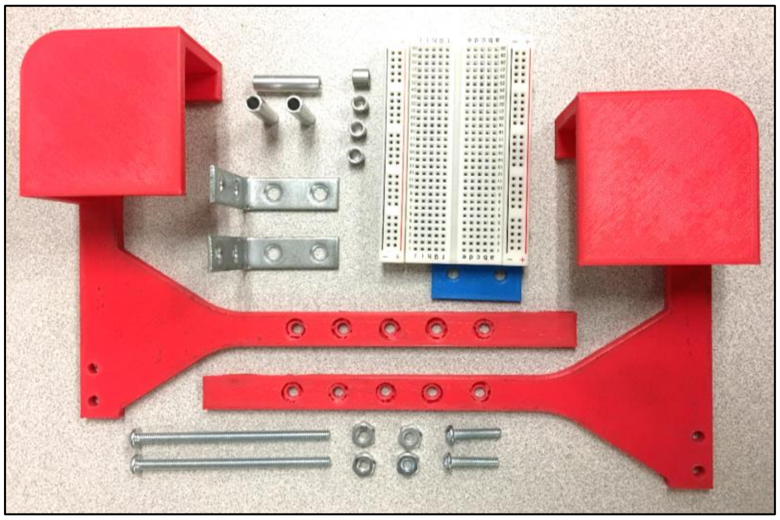


Fig. 53. Baxter mount parts

To increase the bend in each L-bracket, one end of the bracket was secured in a vice as shown in Fig. 54. The bend was then hit lightly with a mallet until the final angle was approximately 85°.

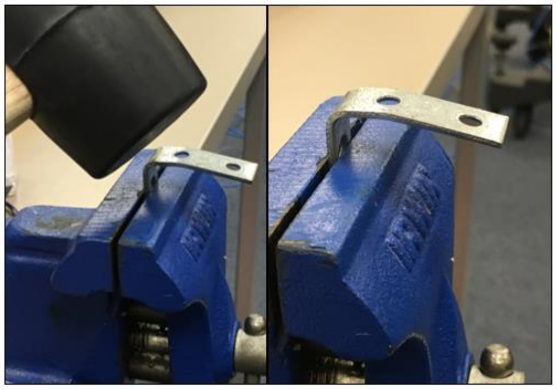


Fig. 54. Bending the L-brackets

One L-bracket was attached to the plate behind the right eye using the two short #8-32 screws and nuts. After sticking the breadboard onto the blue plate, two three-inch #8-32 screws were slid through the breadboard plate's holes. Spacers that added up to two-inches in length were added onto each screw. Fig. 55 shows that three one-inch spacers and four quarter-inch spacers were used. The long screws were then placed through the main plate and second L-bracket and secured with nuts.

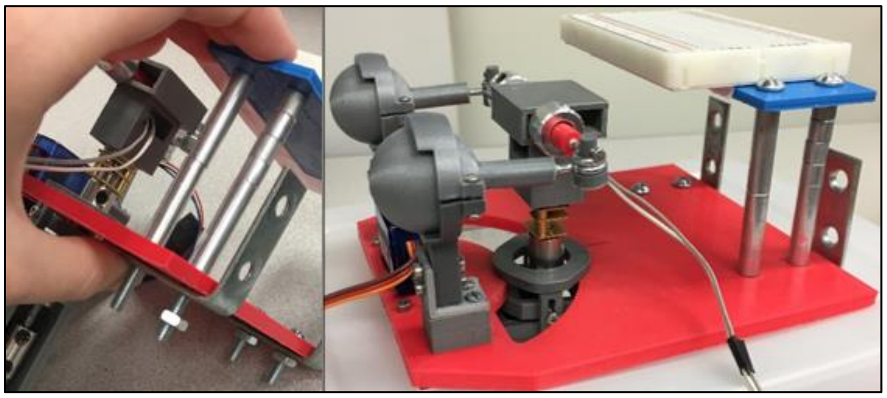


Fig. 55. L-brackets and breadboard plate

Next, wires that connected to the variable power supply's terminals and the wires connected to the DC motor were secured to the motor driver on the Arduino board as shown in Fig. 56.

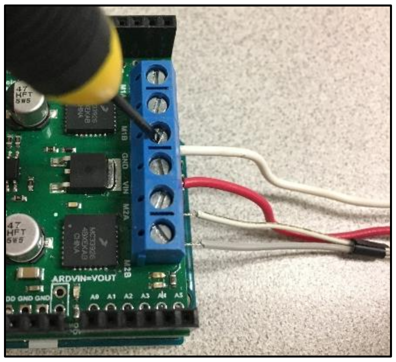


Fig. 56. Connecting motor and power supply to the motor driver

The Arduino was placed on the main plate after covering the two shorter screwheads it was sitting on with electrical tape. All on board wiring was completed as shown in Fig. 57.

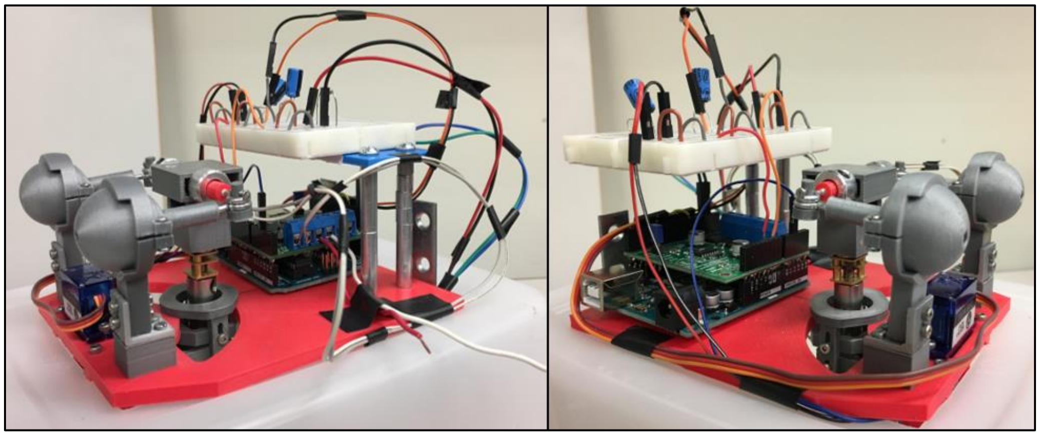


Fig. 57. Wiring

The two mounting supports were secured to the L-brackets using two #8-32 screws and nuts each. Recall that the bracket from the design that secured the two mounting supports to one another is optional. Next, the supports were slid over the top corners of Baxter’s monitor as shown in Fig. 58. The mask was then placed over the eyes and secured by Velcro that was wrapped around the back of Baxter’s monitor and looped through and fastened at both ears.

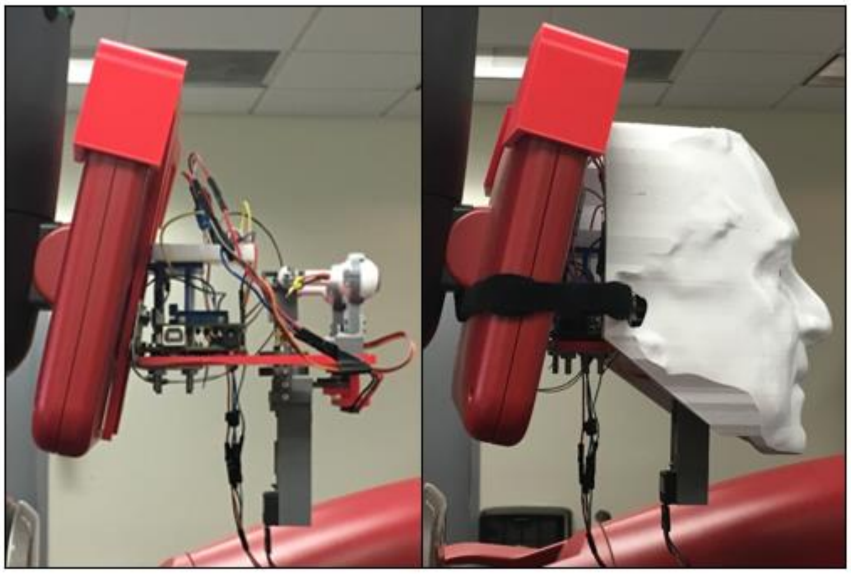


Fig. 58. Mounted assembly

It was discovered that the L-brackets had not been bent enough for the plate to be perfectly level. While bending the L-brackets further was a viable option, the plate was made level by adding a washer to the bottom screws of both brackets as shown in Fig. 59.

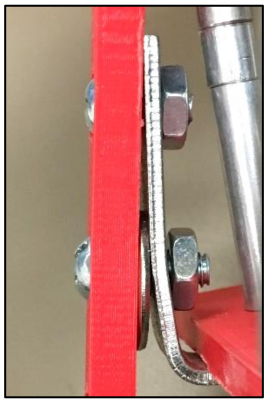


Fig. 59. Washer added to L-bracket

The full assembly is shown in Fig. 60 with and without the laser diode modules powered on.

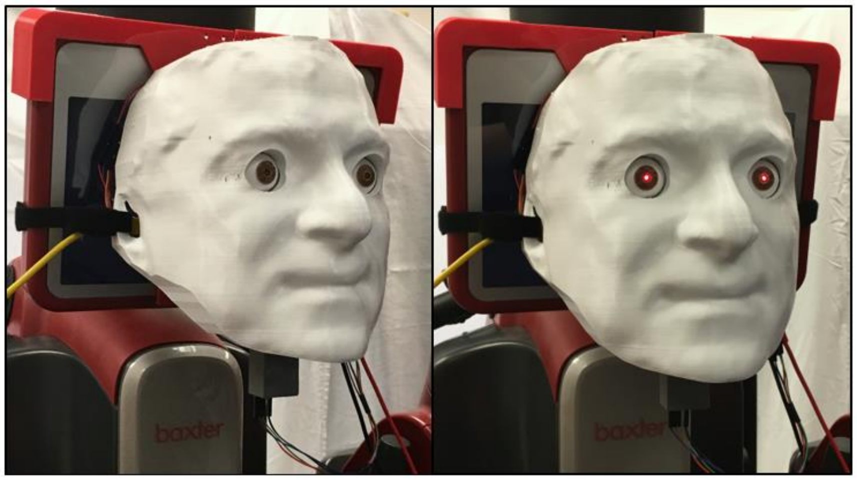


Fig. 60. Full mounted assembly

4.4 Bill of Materials

Table 2 lists the cost and quantity of parts required in the assembly. The only difference between the table and the components used for testing is the power supply. As previously stated, a simple 12V battery can power the DC motor if the application does not warrant a variable power supply. The total is just over two-hundred dollars with the encoder and motor shield accounting for about half of the cost. Using the optional laser diode modules increases the total cost by thirty-three percent.

Item	Cost / Item	# of Items	Total Cost	Vendor
ENS1J-B20-L00100L Optical Encoder	$ 61.99	1	$ 61.99	Arrow
Pololu Dual MC33926 Motor Driver Shield for Arduino	$ 29.95	1	$ 29.95	Pololu
Arduino Uno R3	$ 22.00	1	$ 22.00	Arduino
12VDC Battery	$ 14.99	1	$ 14.99	Home Depot
ROB-12408 Gearmotor 4900 RPM 12VDC	$ 12.95	1	$ 12.95	Digi-Key
10-24 Press-In Nut	$ 6.88	1	$ 6.88	McMaster-Carr
SER0039 9G Metal Gear Micro Servo 1.8kg	$ 5.90	1	$ 5.90	Digi-Key
Breadboard	$ 5.00	1	$ 5.00	Adafruit
Shaft Couplers 3mm-5mm & 1/4in-5Mm	$ 4.99	2	$ 9.98	ServoCity
Potentiometer	$ 2.41	1	$ 2.41	Digi-Key
Shaft Collar 1/4" diameter	$ 2.23	2	$ 4.46	McMaster-Carr
Male-Female Jumper Wires	$ 0.18	7	$ 1.28	Digi-Key
Eyebolt	$ 0.09	4	$ 0.36	McMaster-Carr
Screws, Nuts, Spacers			$ 22.85	Multiple
Misc. (PLA, capacitors, etc.)			$ 14.00	Multiple
		Total	**$ 215.00**	
Laser diode modules (optional)	$ 35.48	2	$ 70.96	Digi-Key
		Total	**$ 285.96**	

Table 2. Bill of materials

5 TESTING: SYSTEM CHECKS

Before testing the final assembly, the locations of Baxter's base and gripper frames were verified, DH parameter calculations were verified, the servo and DC motor were tested individually, and the full assembly was tested using single-point commands. Note that for all testing in this report, it was assumed the eyes were 750mm above Baxter's base frame. When the assembly is mounted to Baxter, the true distance is closer to 650mm. The midpoint between eye centroids is also a few centimeters in front of Baxter's base frame when mounted, not directly above.

5.1 Baxter Gripper Frame Location

Although the location of Baxter's base frame was known, Baxter's PyKDL package did not specify what point on the gripper was being returned by its forward kinematics function. It was found that echoing the topic /robot/limb/left/endpoint_state returned the fingertip location of the electric parallel gripper [38], [39]. For this reason, it was hypothesized that the forward kinematics function also returned the fingertip location.

To test the hypothesis, the forward kinematics function was run for one left gripper position. Without moving the gripper, the endpoint state topic was then echoed. Results for x, y, and z within two-millimeters of one another would prove that the forward kinematics function returned the fingertip location. Otherwise, measurements by hand would be performed to determine the true point location using the fingertip location as a reference.

5.2 DH Parameters

The purpose of this test was to ensure that the equations written in the DH parameter section of code return the correct values for Δx, Δy, and Δz. First, the delta values needed for the eyes to look from straight ahead to a target point were calculated in both Excel and the code. The hypothesis was that the difference between the delta values output by the code and the true values would be less than one millimeter. The one-millimeter limit allowed for negligible rounding errors. Any larger errors would have indicated significant rounding errors caused by breaking the calculation up into too many steps or a typo.

Two sets of target positions were chosen along arcs shown in Fig. 61. In the firsts set, the points were on the arc that contained possible positions of Baxter's right gripper when his fully extended arm moves up and down. The second set was on an arc that contained possible right gripper positions when Baxter's fully extended arm moves left and right. The limitations of Baxter's arm movement were not considered when choosing the twenty-two individual points.

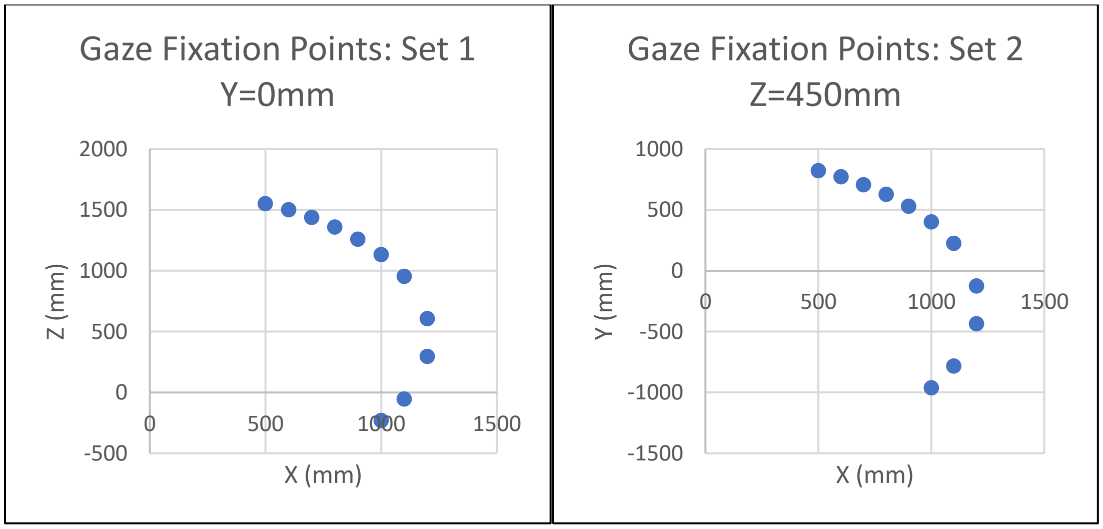


Fig. 61. Points used to verify DH parameters

Matching Excel and Arduino results would rule out rounding errors and typos in the code, but it would not guarantee that the equations themselves were correct. Therefore, the next step was to test the calculations geometrically. Working backwards in Excel, Points A and B in Fig. 62 were derived from the chosen Δz values -13mm and +13mm, respectively. Recall that {1} is the center point between the eyes.

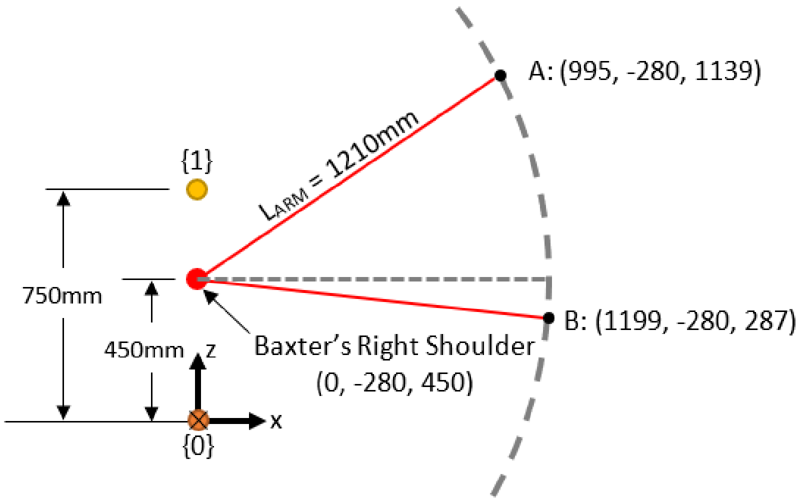


Fig. 62. Points A and B chosen to produce Δz = ±13mm

Fig. 63 is a top-down view of Baxter. The value of M, or the distance between {1} and a point, was found for both A and B using the Pythagorean Theorem.

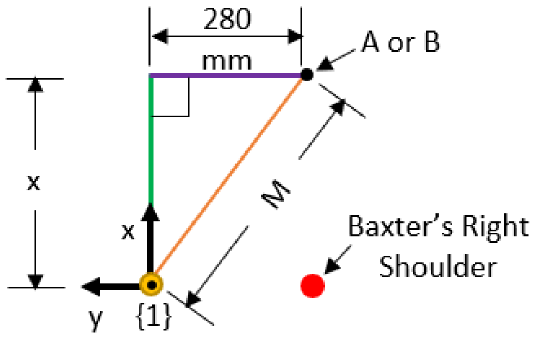


Fig. 63. Top-down view of Baxter for calculations

After finding M_A and M_B, the tilt angles θ_A and θ_B shown in Fig. 64 were calculated using tangent functions. In theory both angles should equal the angle of tilt of the eyeballs create by $\Delta z = \pm 13$mm, which is 20.6°.

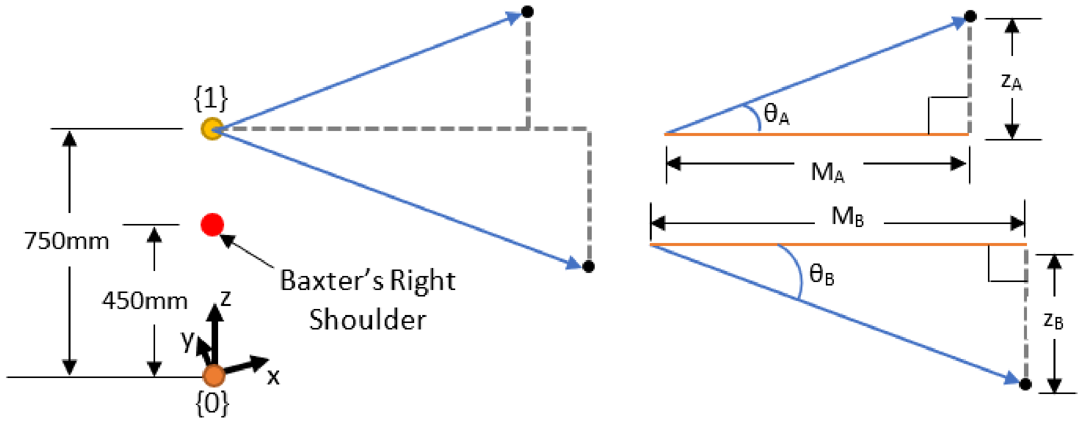


Fig. 64. Calculating θ_A and θ_B; both should equal 20.6°

The second geometric verification used two new points shown in Fig. 65 along the same arc as the second set of target positions. The value of Δy associated with Point C is 20mm and that of Point D is -20mm. Using simple trigonometry, θ_C and θ_D are 34.3° and 33.6°, respectfully.

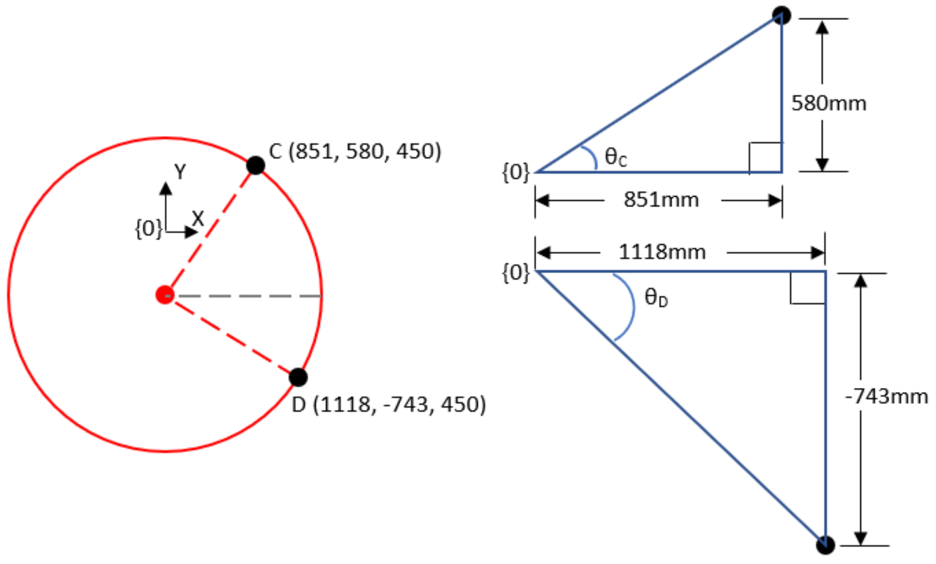


Fig. 65. Points C and D chosen to produce $\Delta y = \pm 20$mm

In Excel, the values of Δz for Points C and D were found to be 10.35mm and 8.07mm, respectfully. Note that the values of Δz were not equal even though both points were the same vertical distance from the base frame. Equations 30 and 31 were used to find θ_C and θ_D for both points. The results should match those found through the simple trigonometry from Fig. 65.

$$r_{xy} = \sqrt{r^2 - z_3^1} \tag{30}$$

$$\theta = \sin^{-1}\left(\frac{20mm}{r_{xy}}\right) \tag{31}$$

5.3 Servo

The servo was put through two sets of tests: one without potentiometer feedback (open-loop control) and one with feedback (closed-loop control). Each set consisted of one test disconnected and one test connected to the entire assembly. The hypothesis was that regardless of whether the servo arm was connected to load or not, the servo would have greater accuracy under closed-loop control than under open-loop control.

5.3.1 Open-Loop: Load Disconnected

Fig. 66 shows the setup for the first open-loop test. A piece of blue tape was stuck along the middle of a mousepad with its long edges parallel to two of the pad’s edges. The servo was screwed to the plate, and the plate was propped on its side and set flushed against a side of the pad perpendicular to the long piece of tape. A camera phone was set flushed against the opposite side, ensuring the plate and camera were parallel. The plate was moved so the servo’s shaft was centered above one of the tape’s long edges. Finally, the camera was moved until the edge of tape appeared vertical through the camera lens, which meant the camera was centered in front of the servo's shaft. Smaller pieces of tape were used to mark where to keep the plate and camera.

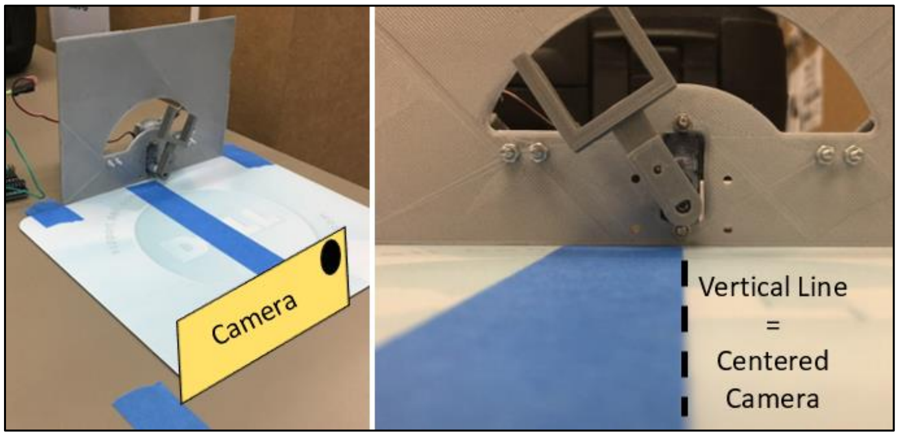


Fig. 66. Disconnected test setup (left); centering the camera (right)

The servo "Sweep" example that came with Arduino was used. In the example, the servo is told to "sweep" back and forth from one angle to another. The same two angles were used for all trials that set a total sweep of 70°. Accuracy was determined by how close to this distance the servo swept across at various speeds. The speed of each trial was controlled by changing the servo's delay, or the amount of time in milliseconds the servo was given to move one degree. The larger the delay, the slower the servo rotated.

Five trials were performed with the delay ranging from 5ms to 25ms. In each trial, the servo arm would begin in a position near the lower limit that was not recorded. It would then move back and forth twice, covering two sweeps. Because it would have been difficult to use a protractor, the videos were analyzed in a program called Kinovea. The servo arm's positions at the extremes of each sweep were measured as shown in Fig. 67.

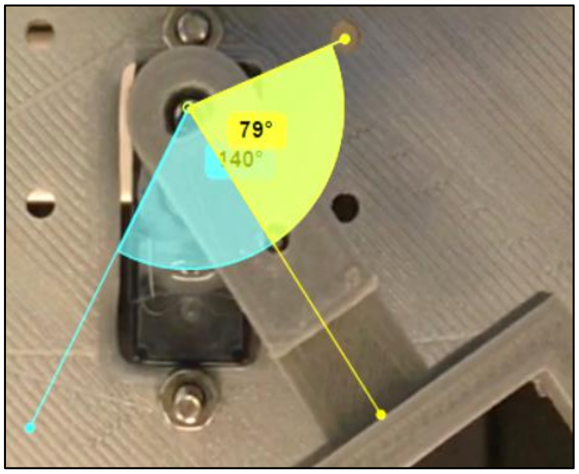


Fig. 67. Measuring angles in Kinovea to calculate the sweep

The measurements from each trial were recorded in Excel as shown in Table 3. The average sweep and its standard deviation were calculated.

Trial 1

Delay=25	**Cycle**	**Start (°)**	**End (°)**	**Sweep (°)**	**SD (°)**
	1		140		
	2	79	141	61	
	3	79		62	
	Average:	79	140.5	61.5	0.71

Table 3. Sample data table for single trial

5.3.2 Open-Loop: Load Connected

With the load connected to the servo arm, the test setup was altered to the one shown in Fig. 68. The bottom and one side of a bucket were cut out, and the bucket was placed upside down on a table. The eye assembly was placed on top, and a camera phone was slid through the side and centered under the servo. The same data collection process from the previous section was used.

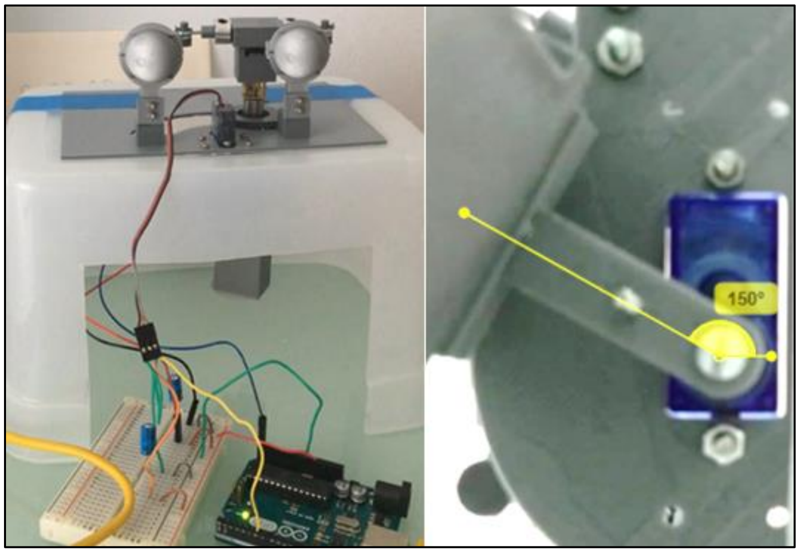


Fig. 68. Connected test setup (left); angle from Kinovea (right)

5.3.3 Closed-Loop: Load Disconnected

The same physical test setup from Section 5.3.1 was used with the addition of the potentiometer, mount, and servo-pot interface piece as shown in Fig. 69.

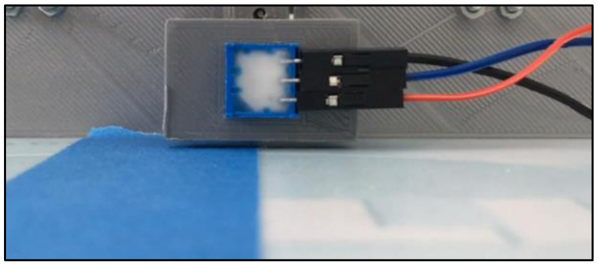


Fig. 69. Potentiometer attached to disconnected test setup

The code, however, did not use Arduino’s Sweep example. Instead, one target angle at a time was written to the servo and the delay remained constant at 10ms, which was the second-lowest delay tested in the no feedback trials. After the servo.write() command was executed, the code calculated the actual angle from the potentiometer’s analog output. If the actual angle differed from the target angle, the servo was instructed to move one degree in the proper direction until the target angle was achieved, creating closed-loop control. The target angles and calculated final angles from the potentiometer were recorded. Note that the code used was not the same as the final code.

Although the angles obtained through Kinovea were also recorded, the data was not analyzed. The Kinovea data would have only been useful if Kinovea angles were also obtained in the connected feedback test described in the coming section. Unfortunately, centering the camera under the assembly with the potentiometer blocking the servo shaft could not be done accurately.

5.3.4 Closed-Loop: Load Connected

The potentiometer, mount and servo-pot interface were added to the previous connected setup, and the code from the other feedback test was used. The target angles and the calculated final angles from the potentiometer were recorded. As previously stated, no analysis was done using Kinovea.

5.4 DC Motor

Three preliminary tests were run with the DC motor. The first tested different encoder codes to ensured that the code chosen accurately tracked the motor’s rotation. The second empirically found the relation between the speed index used by the open source code and motor RPMs. The third tested motor accuracy using slightly different velocity profile.

5.4.1 Encoder Accuracy

Three versions of open source encoder code were tested. The first was a the “Simple Example” from the Arduino rotary encoder playground, the second came from code for a ROS-Arduino Bridge, and the third used the “Interrupt Example” from the Arduino rotary encoder playground [40], [41].

The purpose of this test was to determine what code could accurately track the motor’s rotation at low and high RPMs. Any version that fell short would be removed from the list of options for use in the robotic eye code. The hypothesis was that all three would perform equally, meaning the simplest one would be implemented in the final code.

The same motor code was added to each set of encoder code. Each version was run for two trials at two different speeds: i=400, which is the maximum, and i=100. The former had a run time of T=5000ms and ramp times of 1000ms up and down while the latter had a run time of T=4250ms and ramp times of 250ms up and down.

5.4.2 Speed Index Mapping

The purpose of this test was to find the conversion between the motor’s speed index and RPMs. The hypothesis was that the relation between the two was linear.

After the encoder code was chosen, the same motor code from the previous section was run at ten different maximum speed indices ranging from 85 to 395. For each trial, the time in milliseconds, number of encoder ticks, and speed index were output into the Arduino serial monitor after every loop, and the values were copied into Excel at the end of the trial. The two data points at either end of the trapezoidal velocity profile’s horizontal line were taken and used to calculate the motor’s maximum speed in RPMs for that trial. The process was repeated for all ten trials, and a scatter plot of maximum RPMs versus maximum speed indices was created. The points were curve fit in Excel using various types of equations. The best fit indicated by an R^2 value closest to one and no lower than 0.95 would be implemented in the final code.

5.4.3 Choosing Velocity Profile

A new DC motor velocity profile needed to be calculated in Arduino with every new gaze fixation point. Ideally, the bar centroid would reach its desired vertical position every time an accurate velocity profile was provided; however, there was a chance that factors such as friction would alter performance.

The purpose of this test was to find out if the DC motor performed as expected given a velocity profile free of post-calculation adjustments. In other words, the velocity profile was calculated exactly for the desired Δz. Accurate performance was defined as consistently moving the bar centroid to within ±1mm of its desired vertical position. Proving the hypothesis wrong would lead to testing a slightly modified velocity profile under the same standards.

The initial profile tested was a trapezoidal velocity profile in which the value of Δz determined the maximum velocity of each trial. Three trials had a total run time of T=250 milliseconds and three had T=500 milliseconds. In both cases, twenty percent of the run time was allotted for the ramp up (acceleration) and the same for the ramp down (deceleration) as shown in Fig. 70. A condition was added to the code ensuring the motor reached the trial's maximum velocity at 0.2T.

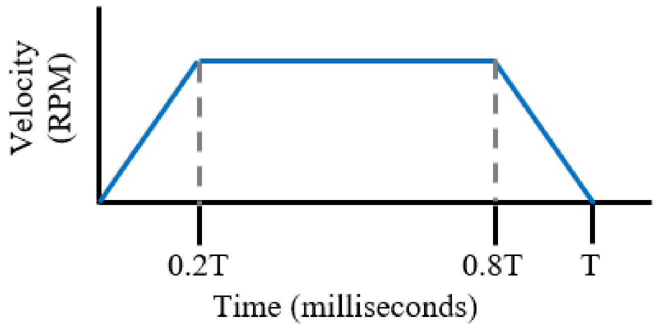


Fig. 70. Velocity profile initially tested

5.5 Lateral and Vertical Motor Movement

Although the final Arduino code used with Baxter gave the motors fixed amounts of time to reach an orientation regardless of saccade size, a test was run to see how the eyes would operate if given a saccade duration based on saccade size. Fig. 71 shows lines A and R created from human eye performance data from two previously mentioned journal papers [6], [9]. The "Test" line was created using the average slope and y-intercept of lines A and R.

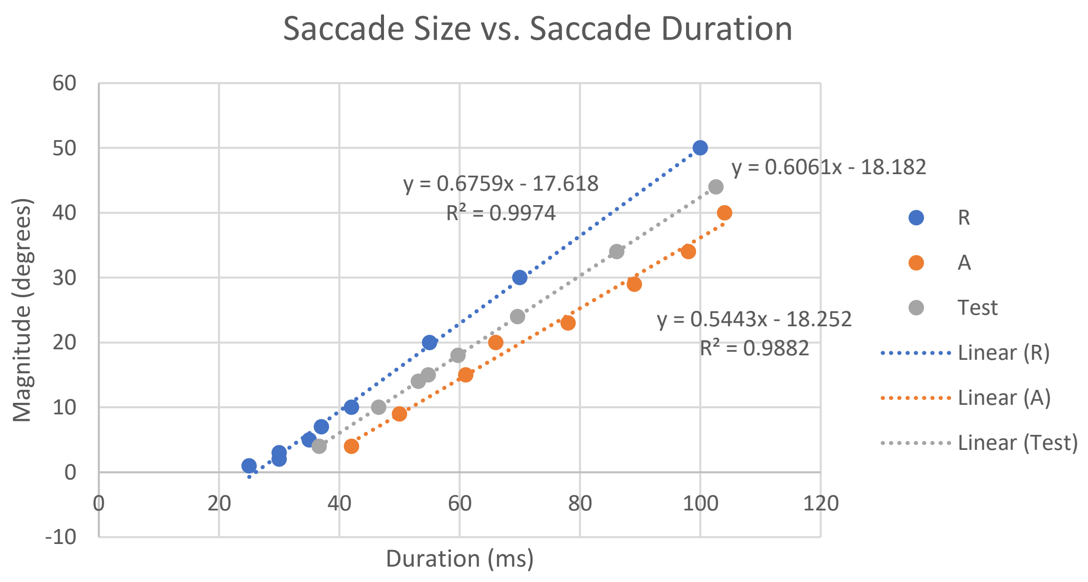


Fig. 71. Saccade size vs. saccade duration

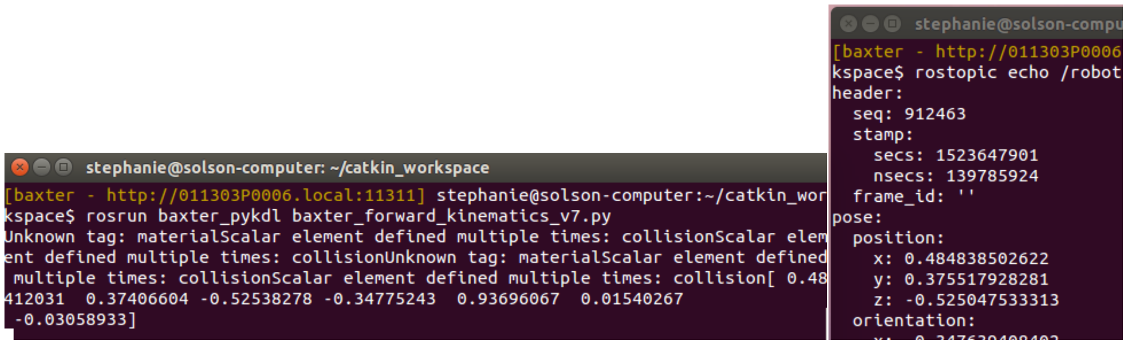
Three sets of tests were performed: one for lateral eye movement, one for vertical eye movement, and one for a combination of the two. A set of points were chosen from the Test line for each test set, and each point was tested in multiple trials for both positive and negative Δθ's. Each trial began with the eyes level and approximately aimed straight forward.

Three versions of Arduino code were used. The code used in the lateral motor test was a subset of the final code that took arguments for time allotted for lateral motor movement and $\Delta\theta$. That of the vertical motor test was also a subset of the final code. It took arguments for time allotted for vertical motor movement and Δz. Finally, the final code (excluding the code required for ROS) was used for the combined motor test. It took two time arguments: one for lateral motor movement and one for vertical motor movement. It also took the X, Y, and Z of a gaze fixation point. The position output returned by each test was in the same form as their inputs (e.g. the vertical test returned the actual Δz obtained, etc.). For the vertical and combined tests, the position inputs and outputs were converted to degrees for analysis. The hypothesis was that the averaged actual saccade sizes of each test set would produce a line that falls somewhere between lines A and R.

6 RESULTS: SYSTEM CHECKS

6.1 Baxter Gripper Frame Location

The results from both methods are shown in Fig. 72. The forward kinematics function returned the point (0.4841, 0.3741, -0.5254), and the end state topic returned (0.4848, 0.3755, -0.5250), both in meters. The components of each point were all well within two millimeters of each other; therefore, it was confirmed that the forward kinematics function returns the gripper's fingertip location.



```
stephanie@solson-computer: ~/catkin_workspace
[baxter - http://011303P0006.local:11311] stephanie@solson-computer:~/catkin_wor
kspace$ rosrun baxter_pykdl baxter_forward_kinematics_v7.py
Unknown tag: materialScalar element defined multiple times: collisionScalar elem
ent defined multiple times: collisionUnknown tag: materialScalar element defined
 multiple times: collisionScalar element defined multiple times: collision[ 0.48
412031  0.37406604 -0.52538278 -0.34775243  0.93696067  0.01540267
 -0.03058933]
```

```
stephanie@solson-compu
[baxter - http://011303P0006
kspace$ rostopic echo /robot
header:
  seq: 912463
  stamp:
    secs: 1523647901
    nsecs: 139785924
  frame_id: ''
pose:
  position:
    x: 0.484838502622
    y: 0.375517928281
    z: -0.525047533313
  orientation:
```

Fig. 72. Baxter gripper frame check results

6.2 DH Parameters

Tables 4 and 5 show the delta values found in the Arduino code and Excel. All data is in millimeters. As hypothesized, the difference never exceeded one millimeter. On average, the values differed by about half of a millimeter. The test proved that the code does not contain any significant rounding errors or typos in the intended equations.

Gripper Movement in Right Shoulder XZ Plane											
Input Point			Arduino Delta			Excel Delta			Difference		
X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
500	-280	1552	18	10	-30	18.23	10.51	-30.10	0.23	0.51	-0.10
600	-280	1501	14	10	-27	14.83	10.35	-27.75	0.83	0.35	-0.75
700	-280	1437	11	10	-24	11.61	10.16	-24.92	0.61	0.16	-0.92
800	-280	1358	8	9	-21	8.62	9.93	-21.57	0.62	0.93	-0.57
900	-280	1259	5	9	-17	5.91	9.67	-17.58	0.91	0.67	-0.58
1000	-280	1131	3	9	-12	3.55	9.37	-12.74	0.55	0.37	-0.74
1100	-280	954	1	8	-6	1.71	8.98	-6.54	0.71	0.98	-0.54
1200	-280	605	1	8	4	1.21	8.35	4.32	0.21	0.35	0.32
1200	-280	295	3	7	12	3.20	7.89	12.82	0.20	0.89	0.82
1100	-280	-54	7	7	21	7.74	7.45	21.39	0.74	0.45	0.39
1000	-280	-231	11	7	25	11.10	7.25	25.41	0.10	0.25	0.41
						Average Magnitude			**0.52**	**0.54**	**0.56**

Table 4. Results from first set of target positions

Gripper Movement in Right Shoulder XY Plane											
Input Point			Arduino Delta			Excel Delta			Difference		
X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z
500	822	450	18	-30	11	18.64	-30.18	11.01	0.64	-0.18	0.01
600	771	450	15	-27	10	15.28	-27.91	10.86	0.28	-0.91	0.86
700	707	450	12	-25	10	12.08	-25.17	10.68	0.08	-0.17	0.68
800	628	450	9	-21	10	9.09	-21.91	10.47	0.09	-0.91	0.47
900	529	450	6	-18	10	6.34	-18.02	10.22	0.34	-0.02	0.22
1000	401	450	3	-13	9	3.92	-13.27	9.92	0.92	-0.27	0.92
1100	224	450	1	-7	9	1.97	-7.13	9.55	0.97	-0.13	0.55
1200	-125	450	1	3	8	1.29	3.72	8.93	0.29	0.72	0.93
1200	-435	450	3	12	8	3.14	12.28	8.47	0.14	0.28	0.47
1100	-784	450	7	20	8	7.59	20.96	8.02	0.59	0.96	0.02
1000	-961	450	10	25	7	10.93	25.06	7.82	0.93	0.06	0.82
						Average Magnitude			**0.48**	**0.41**	**0.54**

Table 5. Results from second set of target positions

The values of M_A, M_B, z_A and z_B calculated from the first geometric test are shown in Fig. 73. θ_A and θ_B were then found to both equal 20.6°, which was the exact angle calculated for $\Delta z = \pm 13$mm. The test proved that the equations for Δz were correct.

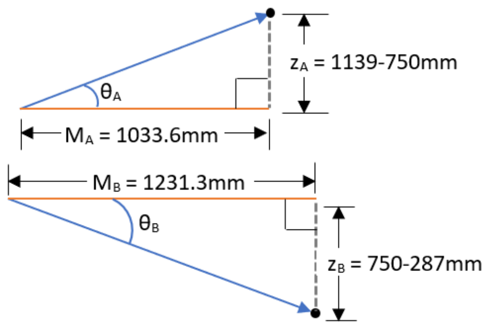


Fig. 73. Calculating θ_A and θ_B

The second geometric test resulted in θ_C and θ_D equal to their previously calculated values of 34.3° and 33.6°, respectfully. The results prove the equations used to find Δy and the servo angle were correct. Because the method used to find Δx was identical to that of Δy and Δz, it was concluded that the equations for Δx were correct as well.

6.3 Servo

6.3.1 Open-Loop: Load Disconnected

The results for the first open-loop test are shown in Fig. 74. The average sweep for every trial fell short of the instructed 70° by at least 9.5°. The data suggested that smaller delays resulted in narrower sweeps, which made sense because smaller delays meant the servo was given less time to move.

Delay (ms)	Sweep (°)	SD (°)
25	61.5	0.71
20	62.0	1.41
15	61.0	0.00
10	58	0.00
5	54.5	0.71

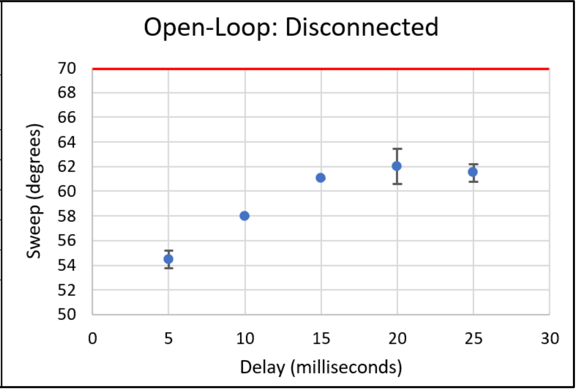
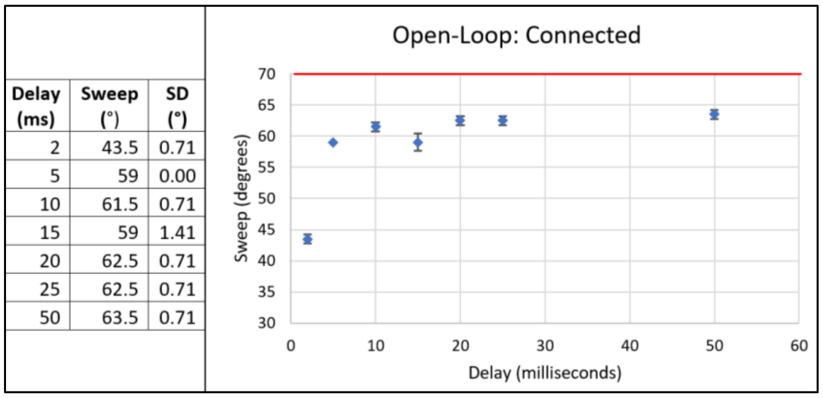


Fig. 74. Results from servo test open-loop: disconnected

6.3.2 Open-Loop: Load Connected

Fig. 75 shows the results for the second open-loop test. The same five delays as before were analyzed as well as delays of 2ms and 50ms.



Delay (ms)	Sweep (°)	SD (°)
2	43.5	0.71
5	59	0.00
10	61.5	0.71
15	59	1.41
20	62.5	0.71
25	62.5	0.71
50	63.5	0.71

Fig. 75. Results from servo test open-loop: connected

Again, smaller delays resulted in narrower sweeps, but the two new delays gave additional information. Decreasing the delay from 5ms to 2ms resulted in a drastically narrower sweep, but doubling the delay from 25ms to 50ms did not create a statistically significant difference in the sweep. The following conclusions were reached for the open-loop control: first, servo performance drops with decreases in delay, and it drops more so when the delay is low. Additionally, the servo reaches a maximum sweep with a large enough delay but may never achieve the instructed sweep angle.

6.3.3 Closed-Loop: Load Disconnected

The results of the closed-loop test with no load are shown in Table 6. The ranges calculated from the potentiometer output were always within three degrees of the input range, and the average difference was only 1.36°. It was concluded that the servo was significantly more accurate in closed-loop control than open loop control when disconnected from a load.

Range Index	Input Range (°)	Calculated Range (°)	Error (°)	Range Index	Input Range (°)	Calculated Range (°)	Error (°)
1	70	68	2	**8**	2	1	1
2	70	68	2	**9**	30	29	1
3	70	69	1	**10**	30	32	2
4	35	36	1	**11**	5	6	1
5	20	19	1	**12**	5	5	0
6	20	19	1	**13**	10	13	3
7	2	2	0	**14**	10	13	3
						Average	**1.36**

Table 6. Results from servo test closed-loop: disconnected

6.3.4 Closed-Loop: Load Connected

Table 7 shows the results of the second closed-loop test. The average error was 2.5° even though the maximum error was six degrees. Although the average and maximum errors were larger than those of the disconnected closed-loop test, the results are significantly better than those from the open-loop connected test. As expected, the servo was more accurate in closed-loop control than open-loop control when connected to the load.

Range Index	Input Range (°)	Calculated Range (°)	Error (°)	Range Index	Input Range (°)	Calculated Range (°)	Error (°)
1	70	70	0	**8**	2	0	2
2	70	70	0	**9**	30	31	1
3	70	71	1	**10**	30	24	6
4	35	32	3	**11**	5	4	1
5	20	21	1	**12**	5	1	4
6	20	14	6	**13**	10	7	3
7	2	0	2	**14**	10	5	5
						Average	**2.50**

Table 7. Results from servo test closed-loop: connected

6.4 DC Motor

6.4.1 Encoder Accuracy

The results of all six trials are shown in Table 8. The simple encoder code had an output of 1 for i_{max}=100 and -218 for i_{max}=400. Knowing the encoder's resolution was 100ticks/rev, the code picked up almost no rotation at the lower speed and just over two rotations at the higher speed over the period of five seconds. It was clearly concluded that the simpler encoder code did not work at the required speeds.

	Trials: Set 1				Trials: Set 2			
Code	i	Run Time (ms)	Ramp Time (ms)	Encoder Ticks	i	Run Time (ms)	Ramp Time (ms)	Encoder Ticks
Simple	400	5000	1000	-218	100	4250	250	1
RAB	400	5000	1000	-124729	100	4250	250	-15493
Interrupts	400	5000	1000	55818	100	4250	250	7068

Table 8. Encoder Accuracy test results

After analyzing the results and conducting further research on encoder code, it was found that the interrupt example code quadrupled the encoder's resolution to 400ticks/rev and the ROS-Arduino Bridge code increased the resolution by a factor of eight. Specifically, each interrupt in interrupt example code doubled the encoder resolution; however, it was not clear how the ROS-Arduino Bridge code achieved a resolution of 800ticks/rev. For that reason, even though the last two sets of code produced comparable results, the interrupt example code was chosen.

6.4.2 Speed Index Mapping

The graph of RPMs versus speed index of the ten trials fitted with a linear curve through the origin is shown in Fig. 76. The line has an R^2 value of 0.8577, which is lower than the minimum 0.95 desired.

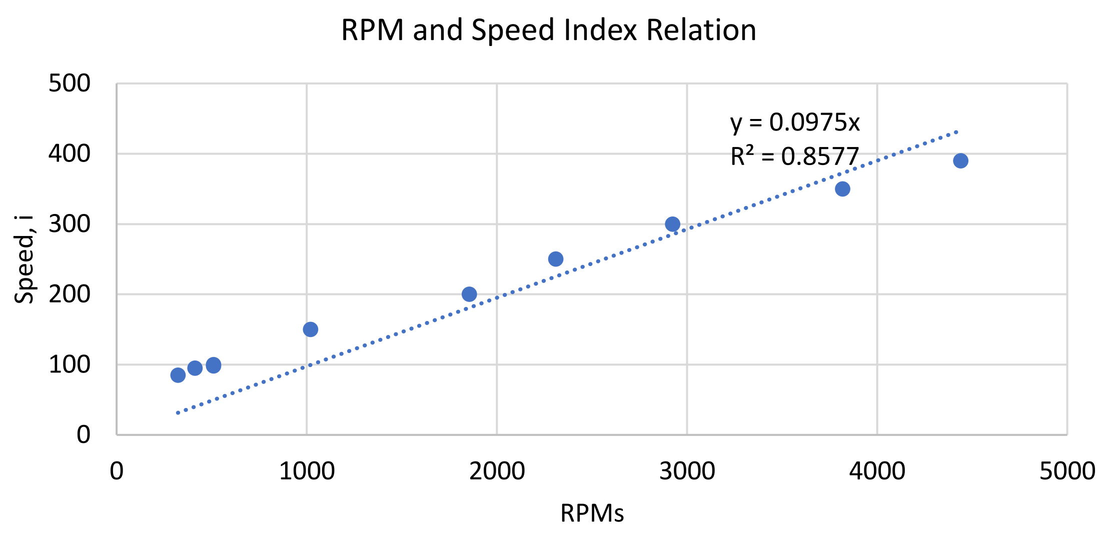


Fig. 76. Initial speed index mapping curve-fit

From analyzing the figure above, it was decided to curve-fit the data without forcing the curve to pass through the origin. After testing multiple types of functions, the polynomial graphed in Fig. 77 was chosen. The line's R^2 value was 0.9977 which was very close to 1, indicating a very good fit to the data.

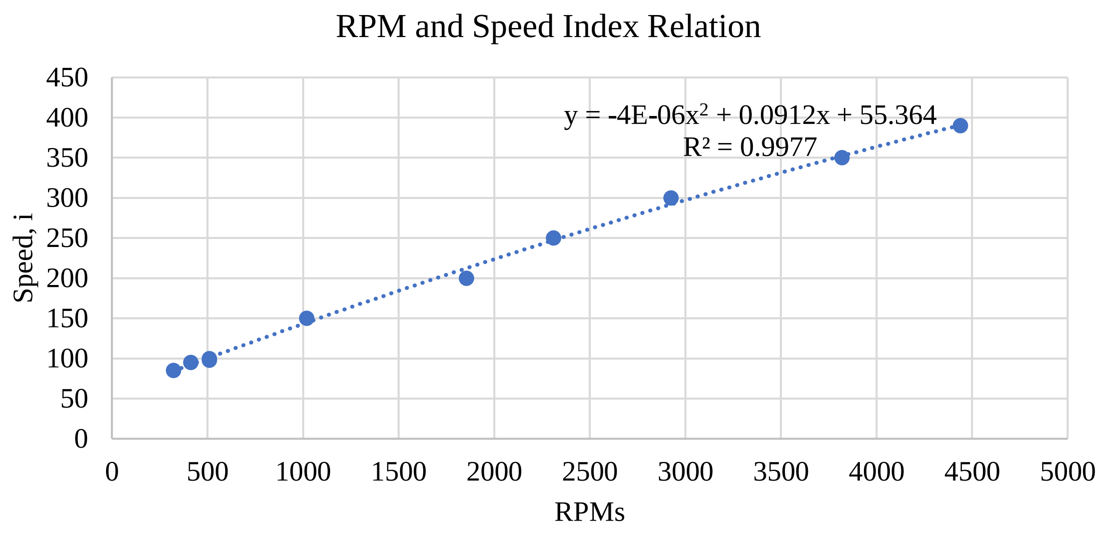


Fig. 77. Final speed index mapping curve-fit

6.4.3 Choosing Velocity Profile

The results of the velocity profile trials are shown in Table 9. The error was greater than one millimeter in all trials, proving the hypothesis false.

Trial #	Run Time (milliseconds)	Δz (mm)	Encoder Ticks	Distance Traveled (mm)	Error (%)	Error (mm)
1	250	-10	3077	-8.14	18.6	1.86
2	250	5	-726	1.92	61.6	3.08
3	250	5	-802	2.12	57.6	2.88
4	500	-10	3241	-8.58	14.2	1.42
5	500	-10	3085	-8.16	18.4	1.84
6	500	-5	45	-0.12	97.6	4.88

Table 9. Results from running ideal velocity profile code

It was clear adjustments needed to be made to improve performance; however, it was discovered after the test that a large part of the issue was mechanical. With a light application of WD-40 to the threaded rod, the vertical motor's performance improved significantly. The following changes were still made to optimize performance. First, a condition was added to the code to check how far the bar centroid had traveled ten milliseconds before the end of the acceleration stage based on encoder output. If it had not travelled as far as expected, a "buffer" speed was calculated and added on to the instructed maximum velocity to make up for lost displacement as shown in Fig. 78. The condition acts as a feed forward control. In addition, feedback control was implemented between the end of the vertical motor's designated runtime and before three-hundred milliseconds had passed. During this time, the vertical motor would run at ±67RPM as it closed in on its desired position.

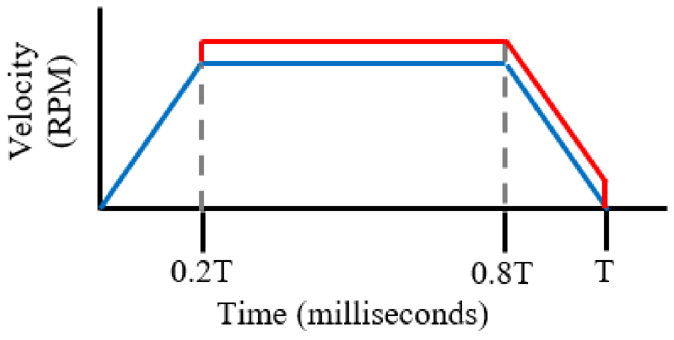


Fig. 78. Coded velocity profile

Eight trials shown in Table 10 were run using the final vertical motor code. All trials were given a run time of 115ms, less than half the time given in the previous trials. The correction condition of the code was accessed after the run time had passed. Δz was the independent variable. The results were a significant improvement from the previous set of trials. The vertical motor was able to obtain its instructed Δz within 350ms in all trials. The motor often overshot its instructed position after obtaining it, but never by more than half of a millimeter. The motor always levelled out within 0.1mm of its instructed Δz, well within the acceptable one-millimeter from the original hypothesis.

Run Time (ms)	Instructed ΔZ (mm)	Time (ms)	ΔZ at Time (mm)	Steady State ΔZ (mm)	Steady State Error (mm)	Steady State Error (%)
115	10	329	10.04	10.05	-0.05	0.5
115	-10	286	-10.04	-9.90	-0.10	1.0
115	7	187	7.02	6.97	0.03	0.4
115	-7	171	-7.05	-6.93	-0.07	1.0
115	4	171	4.02	4.03	-0.03	0.8
115	-4	157	-4.05	-3.97	-0.03	0.7
115	2	174	2.03	2.00	0.00	0.0
115	-2	174	-2.00	-1.93	-0.07	3.5

Table 10. Results from final coded velocity profile

6.5 Lateral and Vertical Motor Movement

The results of the lateral test are shown in Table 11. For a better understanding of what the eyes were doing, a second data point was recorded for each trial at a time after the duration had passed.

Trial	Duration (ms)	Instructed Δθ (deg)	Time 1 (ms)	Δθ (deg)	% Error	Time 2 (ms)	Δθ (deg)	% Error	T2 - T1 (ms)
1	53	14	55	3	**78.6**	117	14	**0.0**	62
2	53	14	55	3	**78.6**	117	13	**7.1**	62
3	53	14	55	1	**92.9**	117	13	**7.1**	62
4	53	14	55	3	**78.6**	117	13	**7.1**	62
5	53	-14	55	1	**107.1**	159	-11	**21.4**	104
6	53	-14	55	0	**100.0**	159	-12	**14.3**	104
7	53	-14	55	0	**100.0**	159	-12	**14.3**	104
8	53	-14	55	1	**107.1**	159	-12	**14.3**	104
9	70	24	72	6	**75.0**	155	24	**0.0**	83
10	70	24	72	5	**79.2**	155	23	**4.2**	83
11	70	24	72	5	**79.2**	155	24	**0.0**	83
12	70	24	72	6	**75.0**	155	23	**4.2**	83
13	70	-24	72	-5	**79.2**	155	-22	**8.3**	83
14	70	-24	72	-6	**75.0**	155	-22	**8.3**	83
15	70	-24	72	-6	**75.0**	155	-23	**4.2**	83
16	70	-24	72	-6	**75.0**	155	-22	**8.3**	83

Table 11. Lateral motor test

It was clear that the eyes were not able to come near their desired position within the instructed durations, the lowest error for that dataset being seventy-five percent. Thus, the hypothesis was proven false. Giving the eyes between one-twentieth and one-tenth of an extra second reduced the error by at least eighty percent and in some cases one-hundred percent.

The reason the eyes could not operate within the tested durations is better understood by graphing the results of all sixteen trials as shown in Fig. 79 and 80. Notice the lateral motor does not even start moving until around thirty and in some cases fiftyfive milliseconds have passed. Therefore, the hypothesis failed not because the lateral motor could not achieve the maximum acceleration to complete the task but rather because motor movement does not begin immediately after the code is uploaded.

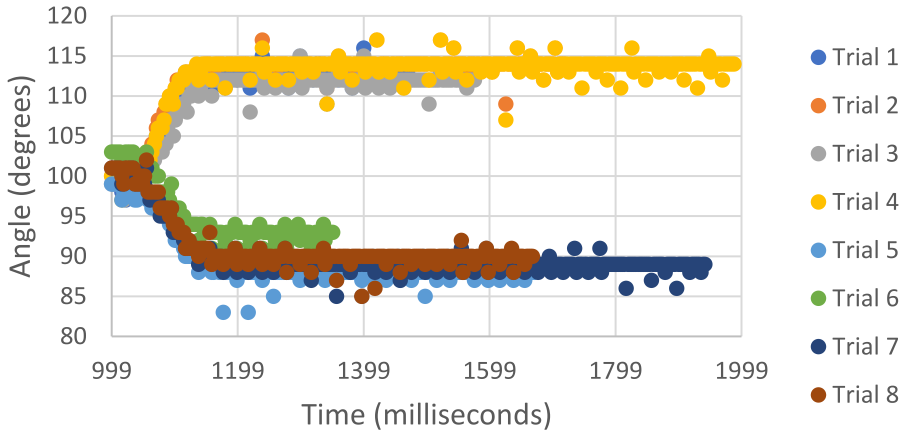


Fig. 79. Lateral motor test: Δθ = ±14°

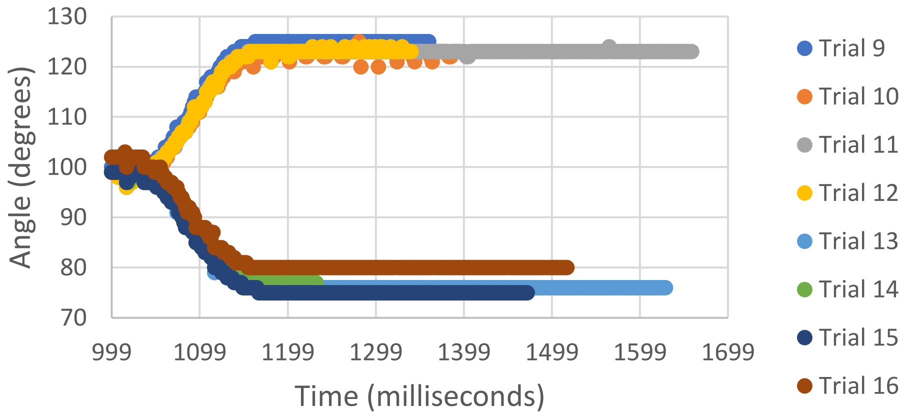


Fig. 80. Lateral motor test: Δθ = ±24°

The results from the vertical motor test are in Table 12. The error was at least seventy-four percent by the end of the given duration in all cases, reflecting the same delay in movement seen in the lateral motor. Again, the hypothesis was proven false. Unlike in the previous test, however, the vertical motor does not quickly reach its desired position one-tenth of a second later. The reason is that the vertical motor is coded to rotate at a speed index of ±125 after the allotted duration has passed, which is only ±67RPM. If the motor were allotted more time to move before correcting its position with a constant, low RPM, the error would decrease. The statement is proven by the vertical motor's accuracy in the final tests with the Baxter robot.

Trial	Duration (ms)	Inst. ΔZ (mm)	Inst. Δθ (deg)	T1 (ms)	ΔZ (mm)	Δθ (deg)	% Error	T2 (ms)	ΔZ (mm)	Δθ (deg)	% Error	T2 - T1 (ms)
1	37	-3	-4.7	38	-0.77	-1.2	**74.4**	104	-3.05	-4.7	**1.7**	66
2	37	-3	-4.7	38	-0.67	-1.0	**77.7**	104	-2.97	-4.6	**1.0**	66
3	37	-3	-4.7	38	-0.69	-1.1	**77.0**	104	-2.99	-4.6	**0.3**	66
4	37	-3	-4.7	38	-0.70	-1.1	**76.7**	104	-3.00	-4.7	**0.0**	66
5	37	3	4.7	38	0.74	1.1	**75.4**	109	3.08	4.8	**2.7**	71
6	37	3	4.7	38	0.61	0.9	**79.7**	109	2.96	4.6	**1.3**	71
7	37	3	4.7	38	0.71	1.1	**76.4**	109	3.05	4.7	**1.7**	71
8	37	3	4.7	38	0.67	1.0	**77.7**	109	3.00	4.7	**0.0**	71
9	47	-6	-9.3	48	-1.09	-1.7	**81.9**	221	-6.02	-9.4	**0.3**	173
10	47	-6	-9.3	48	-1.08	-1.7	**82.1**	221	-5.97	-9.3	**0.5**	173
11	47	-6	-9.3	48	-1.06	-1.6	**82.4**	221	-5.96	-9.3	**0.7**	173
12	47	-6	-9.3	48	-1.07	-1.7	**82.2**	221	-5.97	-9.3	**0.5**	173
13	47	6	9.3	48	0.94	1.5	**84.4**	250	5.93	9.2	**1.2**	202
14	47	6	9.3	48	1.02	1.6	**83.1**	250	6.00	9.3	**0.0**	202
15	47	6	9.3	48	0.97	1.5	**83.9**	250	5.94	9.2	**1.0**	202
16	47	6	9.3	48	1.06	1.6	**82.4**	250	6.03	9.4	**0.5**	202
17	55	-10	-15.7	56	-1.39	-2.2	**86.3**	416	-9.98	-15.6	**0.2**	360
18	55	-10	-15.7	56	-1.40	-2.2	**86.2**	416	-9.98	-15.6	**0.2**	360
19	55	-10	-15.7	56	-1.31	-2.0	**87.1**	416	-9.92	-15.6	**0.8**	360
20	55	-10	-15.7	56	-1.38	-2.1	**86.4**	416	-9.96	-15.6	**0.4**	360
21	55	10	15.7	56	1.29	2.0	**87.3**	485	9.97	15.6	**0.3**	429
22	55	10	15.7	56	1.30	2.0	**87.2**	485	9.87	15.5	**1.3**	429
23	55	10	15.7	56	1.98	3.1	**80.4**	485	10.19	16.0	**2.0**	429
24	55	10	15.7	56	1.29	2.0	**87.3**	485	9.90	15.5	**1.0**	429

Table 12. Vertical motor test

The first set of results from the combined motor test are shown in Table 13. As expected after the previous tests, the eyes never travelled as far as instructed within the time allotted. The error was at least seventy-seven percent in each trial.

Duration (ms)	Time (ms)	Instructed Δθ (deg)	Δθ (deg)	Difference (deg)	% Error
53	57	14.1	2.9	11.2	79.36
53	56	14.1	2.7	11.4	81.12
53	56	14.1	3.2	10.9	77.63
53	55	14.1	2.6	11.5	81.23
53	56	14.1	3.9	10.2	72.57
70	72	24.4	4.6	19.8	81.21
70	72	24.4	3.7	20.7	84.98
70	73	24.4	4.1	20.3	83.38
70	71	24.4	4.0	20.4	83.77
70	72	24.4	4.0	20.4	83.48
86	89	34.6	6.8	27.8	80.40
86	89	34.6	6.6	28.0	81.03
86	90	34.6	5.1	29.5	85.12
86	88	34.6	6.7	27.9	80.59
86	88	34.6	5.9	28.7	82.86

Table 13. Combined motor test: data after duration

As was done in the lateral and vertical motor tests, an additional data point was recorded in each trial as shown in Table 14. Given between one-tenth and one-fifth of a second total to move, the eyes were able to obtain within 1.5° of their desired orientation.

Duration (ms)	Time (ms)	Instructed Δθ (deg)	Δθ (deg)	Difference (deg)	% Error
53	113	14.1	13.15	1.0	6.76
53	113	14.1	13.22	0.9	6.22
53	113	14.1	12.59	1.5	10.69
53	114	14.1	13.33	0.8	5.44
53	113	14.1	13.28	0.8	5.81
70	148	24.4	23.78	0.6	2.54
70	148	24.4	22.87	1.5	6.27
70	147	24.4	22.88	1.5	6.22
70	147	24.4	23.78	0.6	2.55
70	149	24.4	23.77	0.6	2.58
86	179	34.6	34.87	-0.3	0.78
86	180	34.6	33.91	0.7	1.99
86	180	34.6	33.95	0.7	1.89
86	180	34.6	33.90	0.7	2.01
86	181	34.6	34.87	-0.3	0.77

Table 14. Combined motor test: data after twice the duration

Although the hypothesis was proven false, the analysis justified the decisions made for the final robotic eye code implemented with Baxter.

7 TESTING: BAXTER INTEGRATION

7.1. Circular Path: Tracking

In this test, Baxter's gripper moves in the circular path explained in Section 3.3 while the eyes track the gripper. The hypothesis was the eyes would look on average at a point within a 50mm radius of the instructed point. In addition, the distance between the instructed and actual point, or the error, would never exceed 100mm. Data sets of time, X, Y, Z, Y_{EYES}, and Z_{EYES} were output to text files.

7.2. Parabolic Path: Tracking

Baxter performed the first object manipulation task explained in Section 3.3 for the next three tests. Recall that the task was to pick an object up off a platform, move it to the right along an inverted parabolic path, and set it down. The same hypothesis as that of the previous test was used here with the exception that errors larger than 10cm were expected within the first second of the eyes switching from looking forward to looking at the gripper. The same data sets as those in the previous test were collected.

7.3. Parabolic Path: Block Mode

The second tracking test was mimicked after changing the mode in the Arduino code to block mode. The same hypothesis as previous was tested, expanding the exception to include large errors one second after the start and end of target saccade instructions. To work around the issue of publishing to eight ROS Topics from Arduino, the test was performed twice: once obtaining time, X, Y, Z, Y_{EYES}, Z_{EYES} and once obtaining time, X, Y, Z, Y_{HAND}, Z_{HAND}. The values of Y_{HAND} and Z_{HAND} after the eyes leave the grasp site and before the task is complete are simply added to the first set of results for a comprehensive data analysis. To do this, the data sets are aligned based on the first target fixation, not time.

7.4. Parabolic Path: Full Glass Mode

The testing and data acquisition processes from the block test were repeated for full glass mode. The same hypothesis as previous was used.

7.5. Linear Path

The same hypothesis tested using the parabolic path were tested using the linear path explained in Section 3.3.

8 RESULTS: BAXTER INTEGRATION

The following are the results from all tests performed with the Baxter robot. Butterworth filters were applied to the actual gaze fixation point data sets using MATLAB, and the filtered data was exported to Excel. Analysis was performed in Excel and MATLAB. Note that all error distributions only extend so far as to show at least ninety-five percent of the data. Brackets with labels were used to convey where the eyes were instructed to look throughout each trial. A key for the labels is given in Table 15.

Label	**Instructed Gaze Fixation**
Forward	Straight Ahead
Hand	At the left gripper
Grasp	At the grasp site
T	At the target
O	At a point temporally offset ahead of the left gripper

Table 15. Results label key

8.1. Circular Path: Tracking

The instructed and actual gaze fixation points during the first test are shown in Figures 81, 82, and 84. Recall that the x-coordinate of the gaze fixation point was equated to that of the gripper. When the Arduino code is run, the value of x is its initialized value of 500mm until the gripper's location begins publishing to ROS. At that point, the value of x remains within 1029mm and 1067mm.

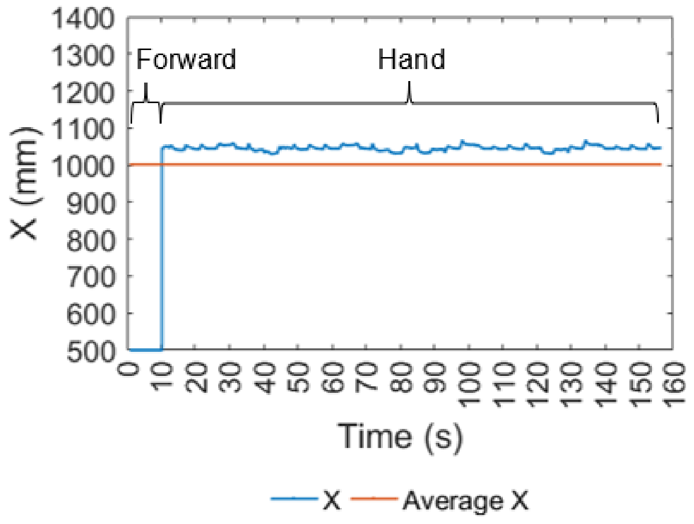


Fig. 81. Test 1: X vs. Time

Fig. 82 plots the y-values of the gripper location and actual gaze fixation point versus time. The average difference between Y and Y_{EYES}, or the error, was 27mm. The error distribution is in Fig. 83. The error fell within 53mm ninety percent of the time and within 70mm ninety-five percent of the time. Overall, tracking from right to left was worse than tracking left to right; however, even the larger errors such as that when Time = 133.4s was only three degrees. In terms of servo angle, the average error was 1.45°.

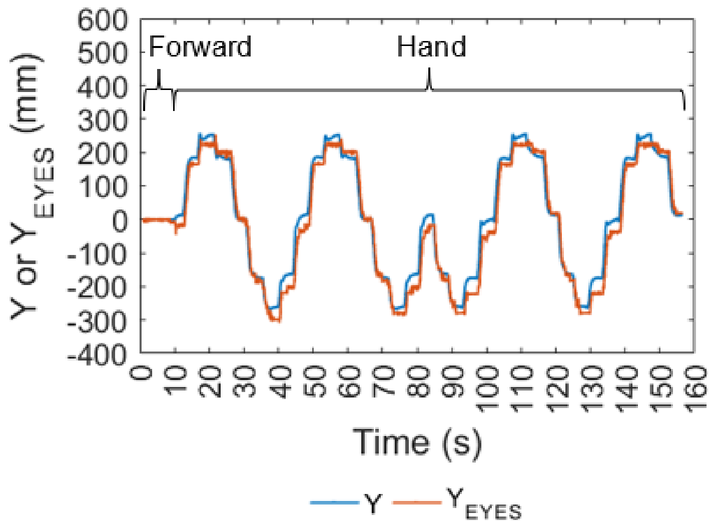


Fig. 82. Test 1: Lateral Motor Analysis

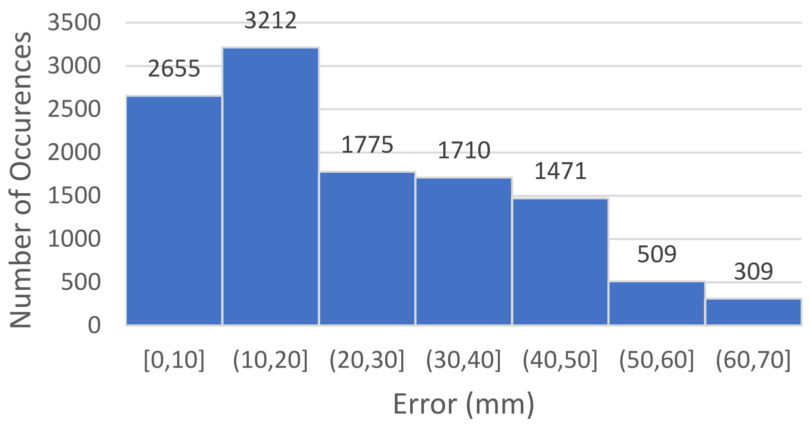


Fig. 83. Test 1: lateral motor error distribution

Fig. 84 plots the z-values of the gripper location and actual gaze fixation point versus time. The eyes tracked better in the z-direction than in the y-direction; the average error was only 5.5mm compared to 27mm. The error distribution is in Fig. 85. The error fell within 15mm ninety percent of the time and within 26mm ninety-five percent of the time.

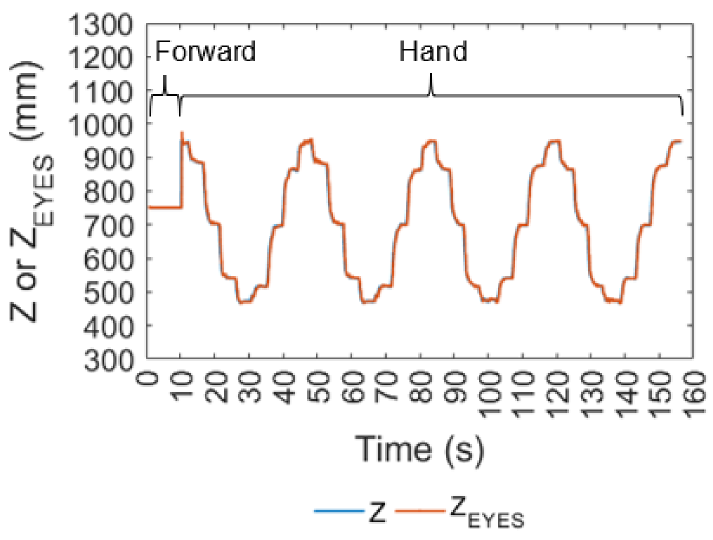


Fig. 84. Test 1: Vertical Motor Analysis

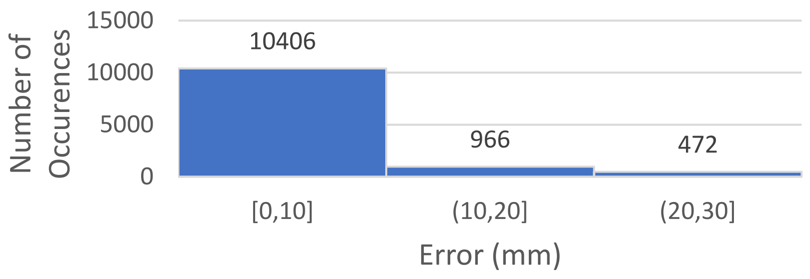


Fig. 85. Test 1: vertical motor error distribution

The distance between the gripper location and gaze fixation point was defined as the magnitude of error shown in Fig. 86. The average magnitude of error was 29mm, which is less than the hypothesized average of 50mm. The first part of the hypothesis was, therefore, proven true. Excluding the spike in error when the eyes were first instructed to track the gripper, five spikes in error exceeded the hypothesized 100mm maximum but were all less than 130mm. Even though the second part of the hypothesis was proven false, the eyes' performance was close to expected. The error distribution is in Fig. 87. The actual gaze fixation point was within 56mm of the instructed point ninety percent of the time and within 72mm ninety-five percent of the time.

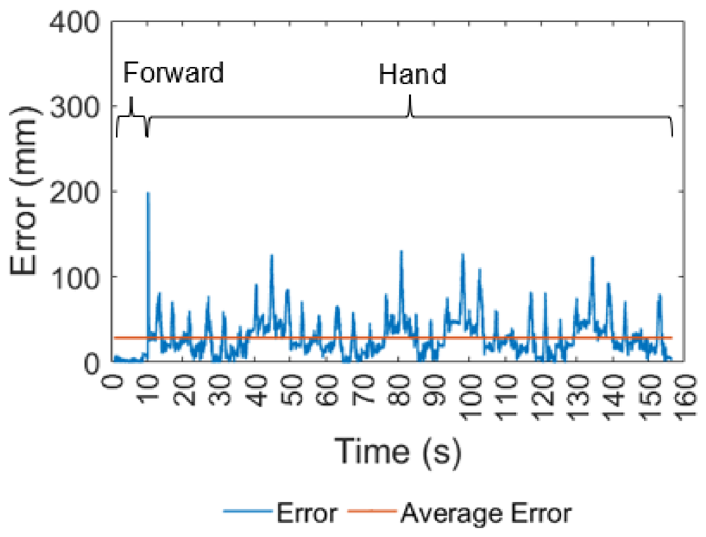


Fig. 86. Test 1: Magnitude of Error

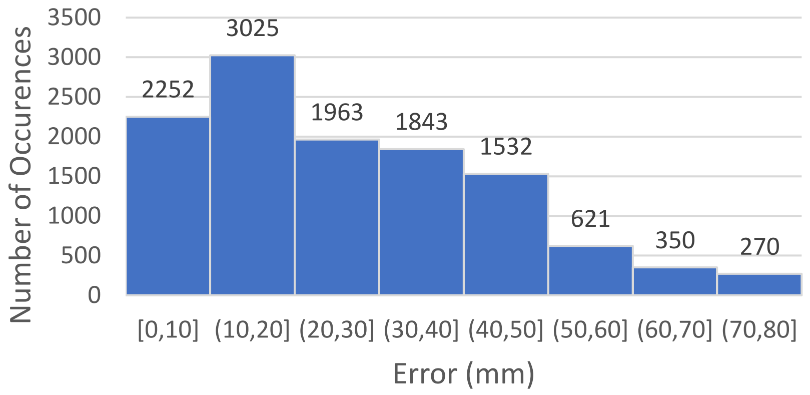


Fig. 87. Test 1: magnitude of error distribution

Tracking in the YZ plane over time is plotted in Figures 88 and 89. In both figures, the data points prior to the gripper location being obtained in Arduino were excluded. For a clearer visual, the first half of the test in which the gripper moved in two clockwise circles and the second half of the test in which the gripper moved in two counterclockwise circles were plotted separately.

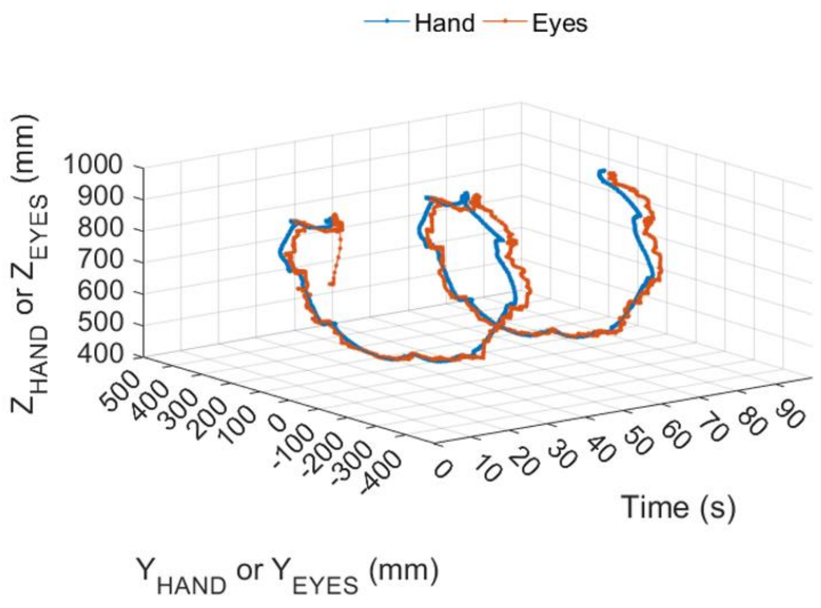


Fig. 88. Test 1 YZ plane clockwise analysis

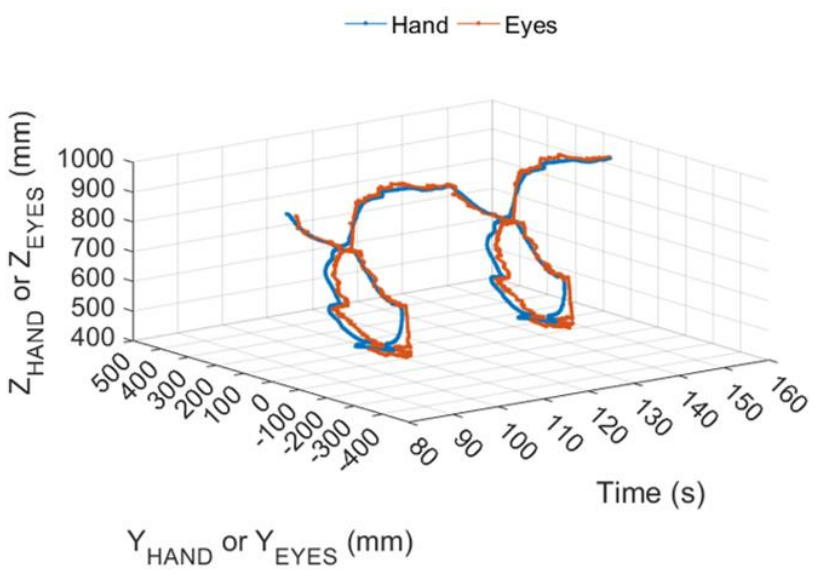


Fig. 89. Test 1 YZ plane counterclockwise analysis

8.2. Parabolic Path: Tracking

The instructed and actual gaze fixation points during Test 2 are shown in Figures 90, 91 and 93. The gripper position began publishing to ROS about 16.5 seconds after the Arduino code was started, at which point the eyes began tracking the gripper.

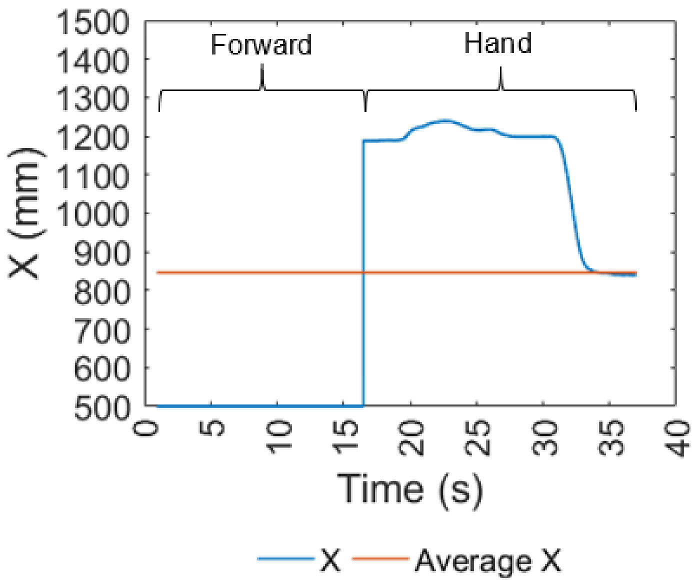


Fig. 90. Test 2: X vs. Time

Fig. 91 plots the y-values of the gripper location and actual gaze fixation point versus time. The steady-state error after the eyes began to follow the gripper but before the gripper began to move right was about 30mm. The average error for the entire track was 29mm. The error distribution is in Fig. 92. The error fell within 59mm ninety percent of the time and within 79mm ninety-five percent of the time. As the gripper moved left to right, the eyes lagged behind the gripper in the y-direction by between half and one second, accounting for most of the error.

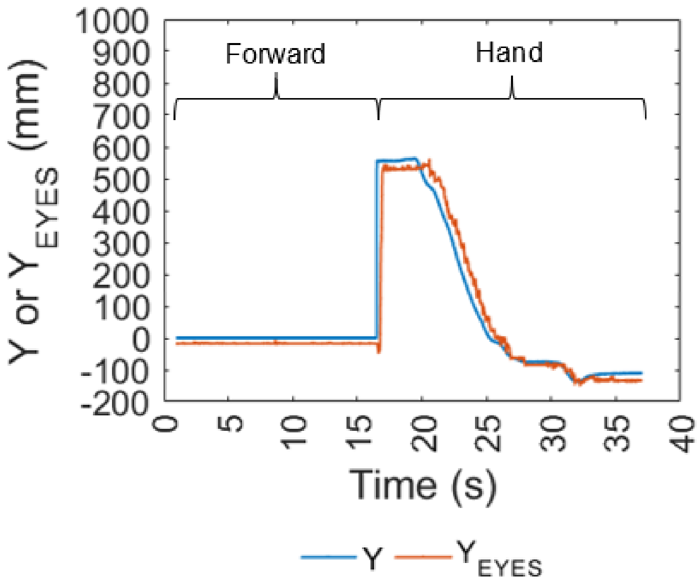


Fig. 91. Test 2: Lateral Motor Analysis

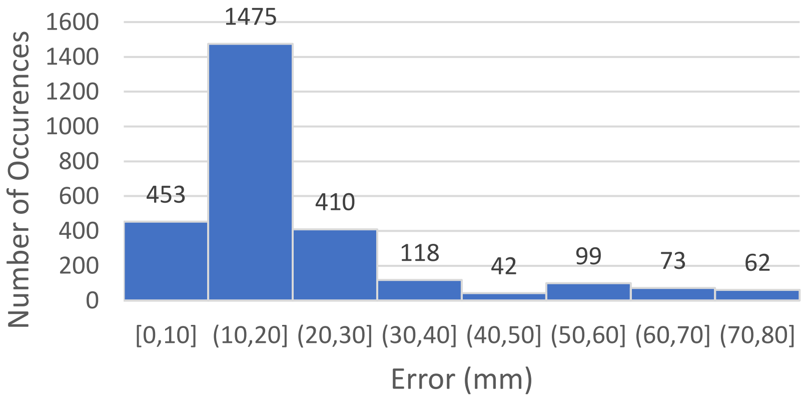


Fig. 92. Test 2: lateral motor error distribution

Fig. 93 plots the z-values of the gripper locations and actual gaze fixation points versus time. Again, the eyes tracked better in the z-direction than in the y-direction; the average error was 4.5mm compared to 29mm in y. The error distribution is in Fig. 94. The error fell within 9mm ninety percent of the time and within 13mm ninety-five percent of the time.

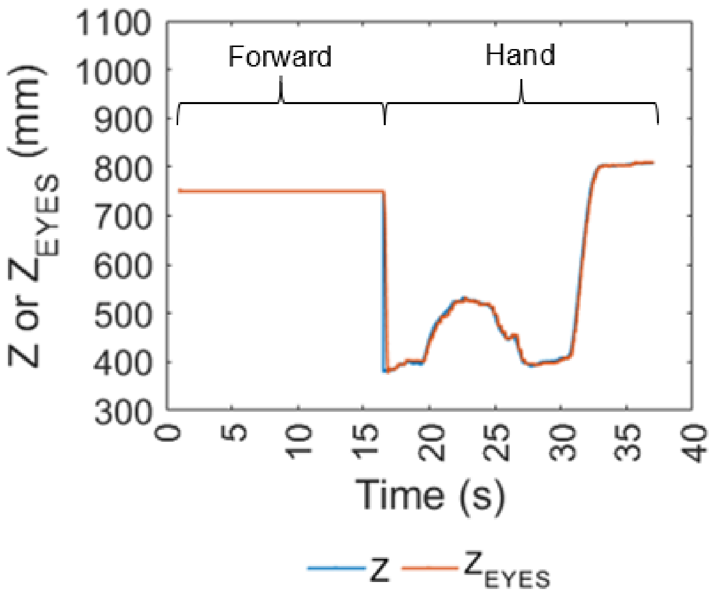


Fig. 93. Test 2: Vertical Motor Analysis

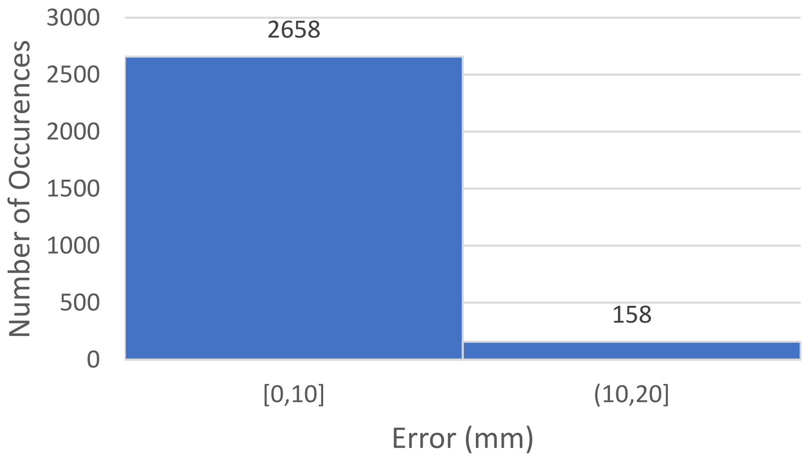


Fig. 94. vertical motor error distribution

The magnitude of error is shown in Fig. 95. The average error was 30mm, which is less than the hypothesized 50mm. Excluding the spike in error when the eyes are first instructed to track the gripper, the largest magnitude of error was 106mm, just over the hypothesized 100mm maximum. The two statistics are very close to those from Test 1.

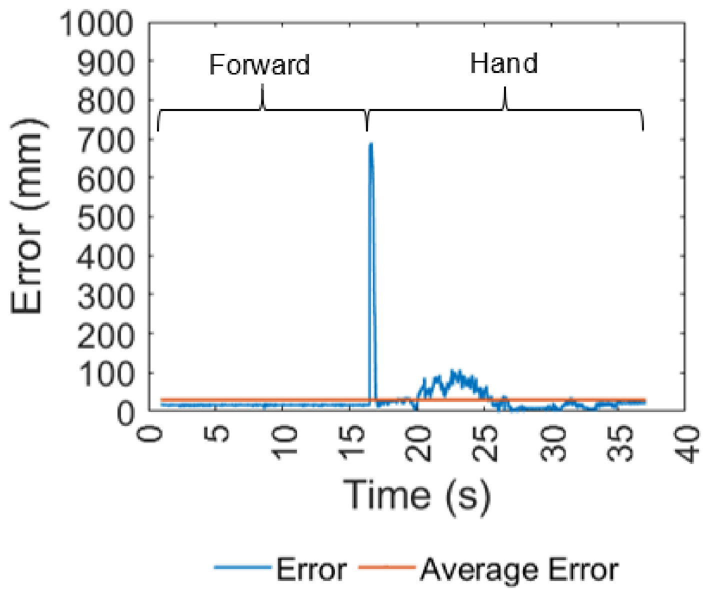


Fig. 95. Test 2: Magnitude of Error

The error distribution is in Fig. 96. The actual gaze fixation point was within 59mm of the instructed point ninety percent of the time and within 79mm ninety-five percent of the time.

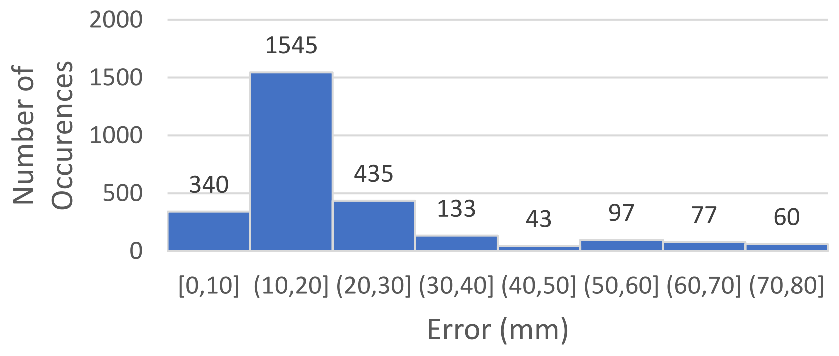


Fig. 96. Test 2: magnitude of error distribution

8.3. Parabolic Path: Block Mode

The instructed and actual gaze fixation points during Test 2 are shown in Figures 97, 98, and 100. The gripper position began publishing to ROS about 4.6 seconds after the Arduino code was started, at which point the eyes began looking at the grasp site.

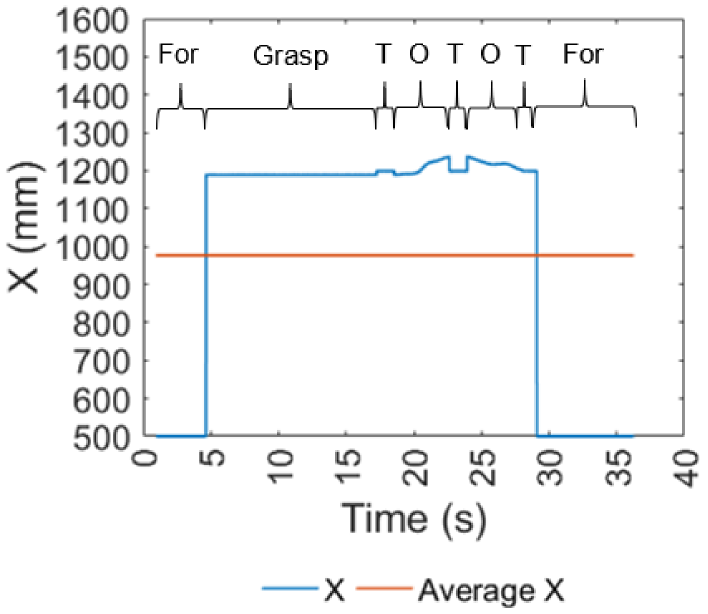


Fig. 97. Test 3: X vs. Time

Fig. 98 plots the y-values of the instructed and actual gaze fixation points versus time. The gripper's y location is also plotted from the first target fixation through the end of the last. The steady-state error while the eyes fixated the grasp site was 22mm. The eyes always looked ahead of the gripper when instructed to during the offset phases; however, there was some delay in achieving those instructed points. The error distribution is in Fig. 99. The average error was 37mm, and the error fell within 36mm ninety percent of the time and within 63mm ninety-five percent of the time.

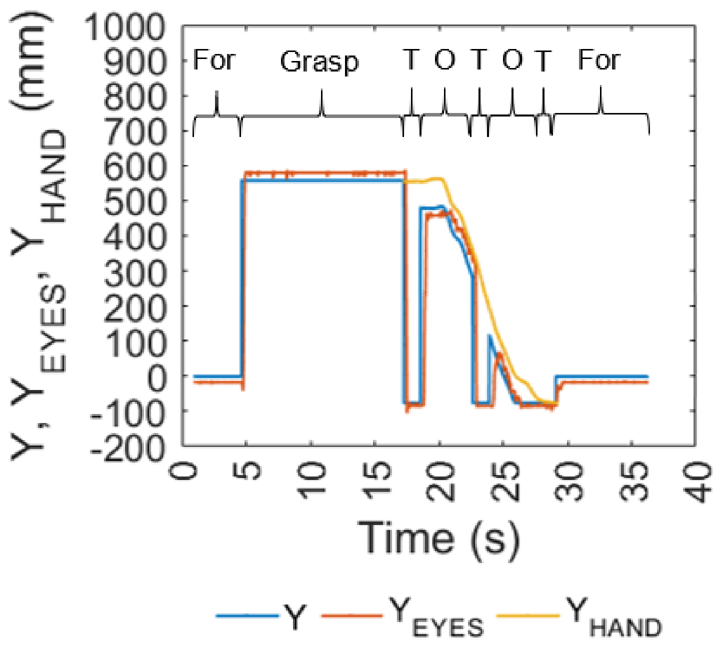


Fig. 98. Test 3: Lateral Motor Analysis

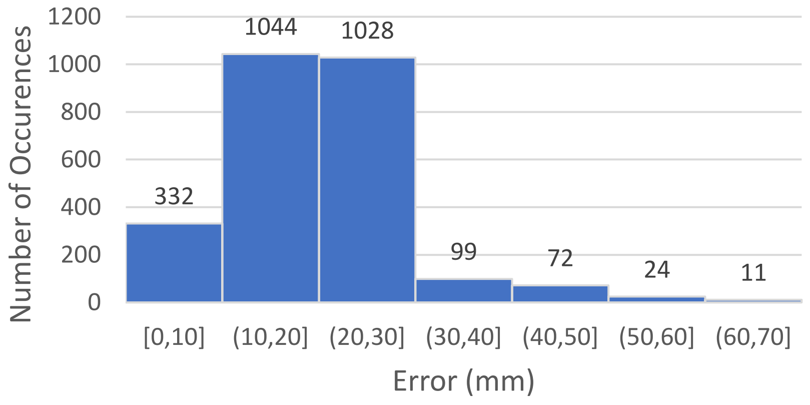


Fig. 99. Test 3: lateral motor error distribution

Fig. 100 plots the z-values of the instructed and actual gaze fixation points versus time. The gripper's z location is also plotted from the first target fixation through the end of the last. The average error was 12mm. Most larger errors occurred directly after abrupt changes in the instructed gaze fixation point. Additionally, for larger saccades such as those to and from the target, the vertical motor briefly overshot its instructed position.

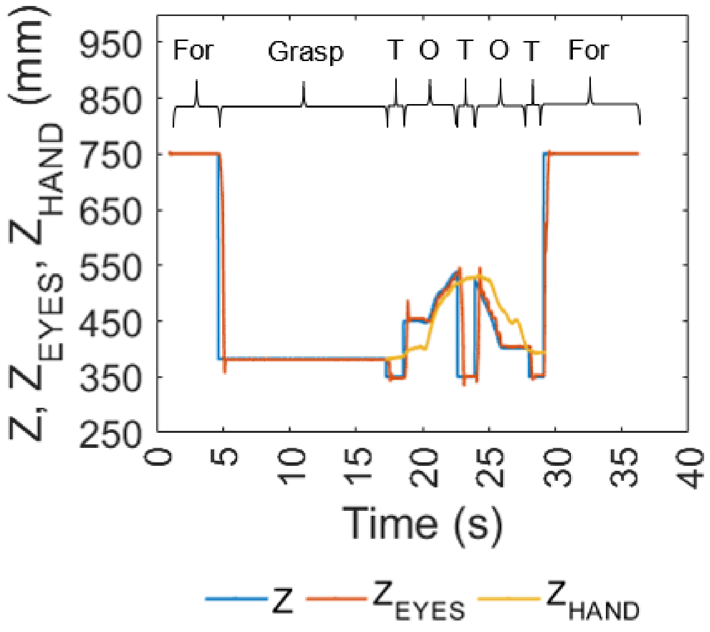


Fig. 100. Test 3: Vertical Motor Analysis

The error distribution is in Fig. 101. The error fell within 13mm ninety percent of the time and within 52mm ninety-five percent of the time.

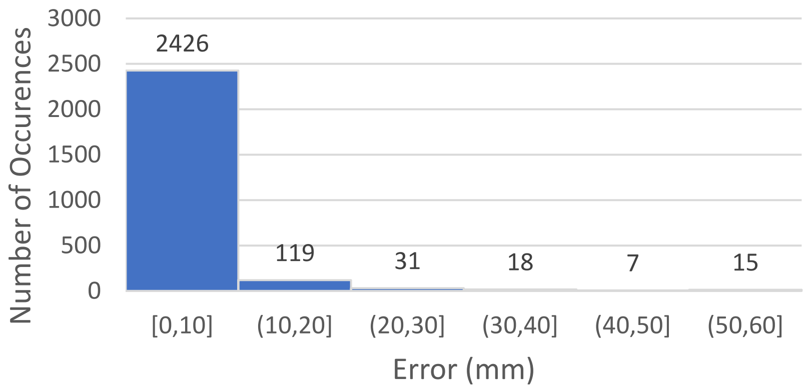


Fig. 101. Test 3: vertical motor error distribution

The magnitude of error is shown in Fig. 102. The average error was 42mm, which is within the hypothesized average error of 50mm. Apart from the spikes in error when gaze instructions changed abruptly, the error never exceeded 100mm for points that occurred at least one second after those spikes. Therefore, the second part of the hypothesis was also proven true.

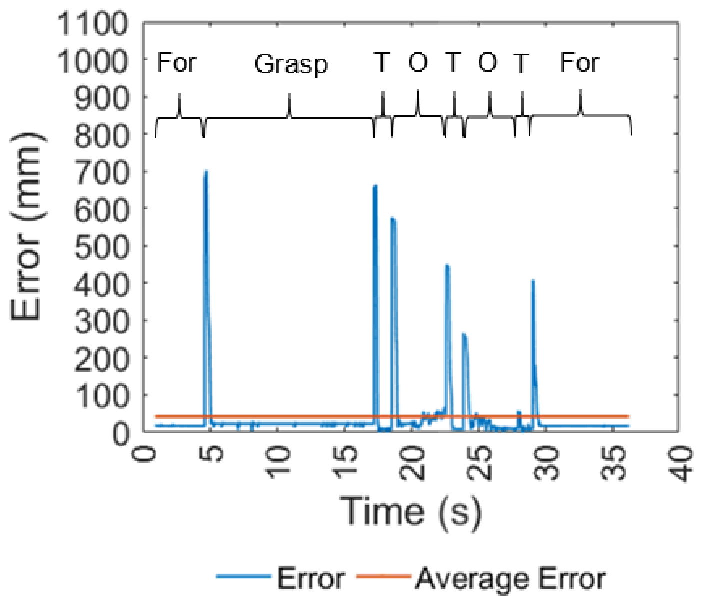


Fig. 102. Test 3: Magnitude of Error

The error distribution is in Fig. 103. The actual gaze fixation point was within 43mm of the instructed point ninety percent of the time and within 162mm ninety-five percent of the time.

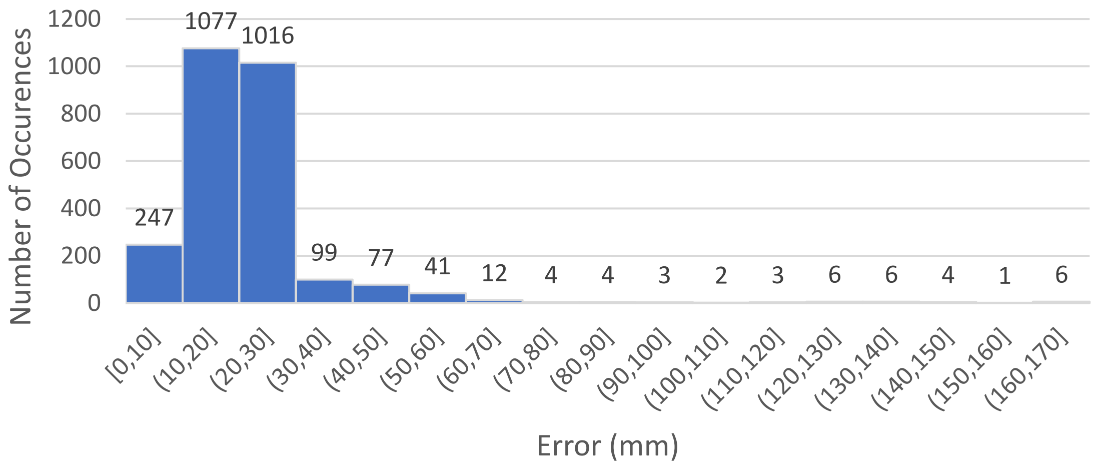


Fig. 103. Test 3: magnitude of error distribution

8.4. Parabolic Path: Full Glass Mode

The instructed and actual gaze fixation points during Test 2 are shown in Figures 104, 105, and 107. The gripper position began publishing to ROS about 4.5 seconds after the Arduino code was started, at which point the eyes began looking at the grasp site.

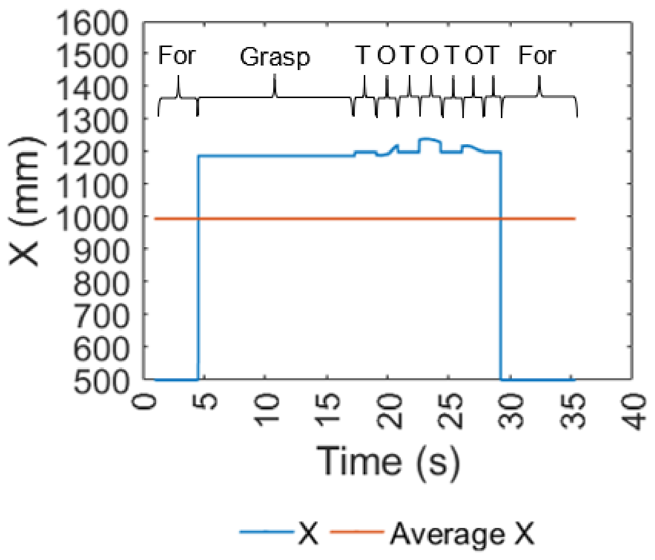


Fig. 104. Test 4: X vs. Time

Fig. 105 plots the y-values of the instructed and actual gaze fixation points versus time. The gripper's y location is also plotted from the first target fixation through the end of the last. The average error was 40mm. The steady-state error when the eyes fixated the grasp site was 28mm, equivalent 1.1° in this case. Unlike in the test running block mode, the eyes were not always ahead of the gripper; the delay in movement caused the eyes to look at or slightly ahead of the gripper during parts of the offset phases. Those errors, however, were usually within three degrees and never exceeded four degrees.

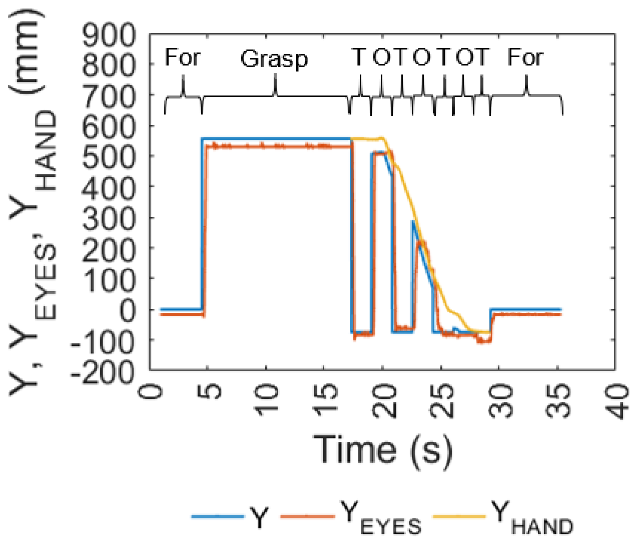


Fig. 105. Test 4: Lateral Motor Analysis

The error distribution is in Fig. 106. The error fell within 35mm ninety percent of the time and within 97mm ninety-five percent of the time.

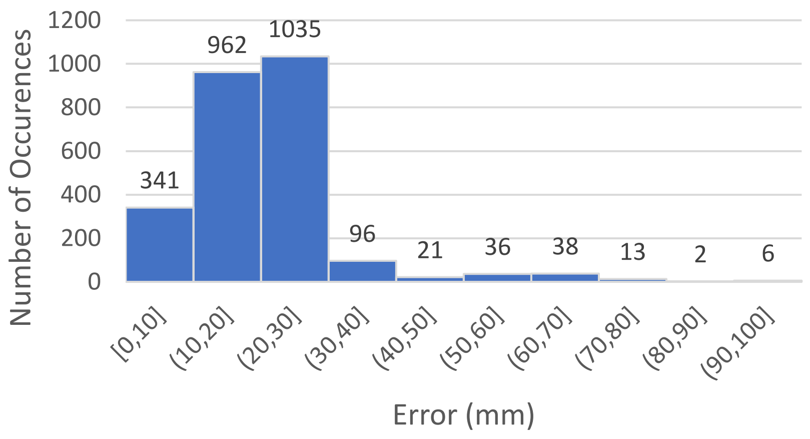


Fig. 106. Test 4: lateral motor error distribution

Fig. 107 plots the z-values of the instructed and actual gaze fixation points versus time. The gripper's y location is also plotted from the first target fixation through the end of the last. The average error was 11mm, 29mm less than that of the lateral motor. Again, the vertical motor briefly overshot its instructed position during the larger saccades.

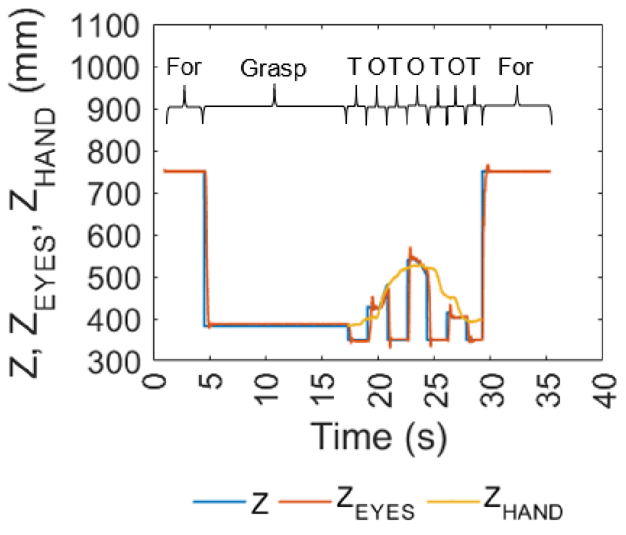


Fig. 107. Test 4: Vertical Motor Analysis

The error distribution is in Fig. 108. The error fell within 11mm ninety percent of the time and within 45mm ninety-five percent of the time.

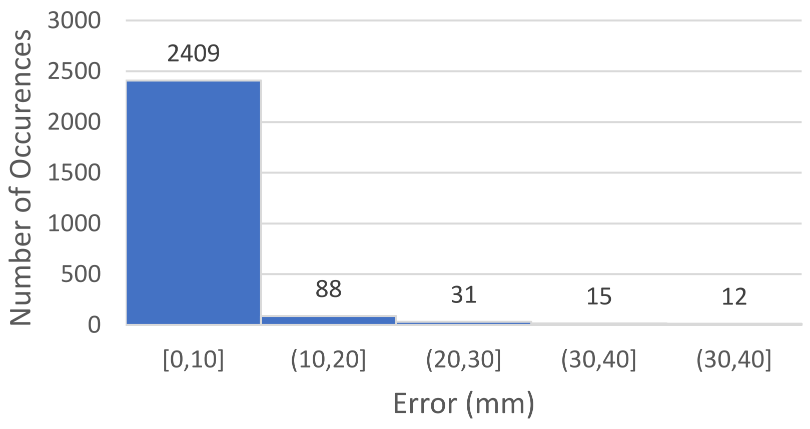


Fig. 108. Test 4: vertical motor error distribution

The magnitude of error is shown in Fig. 109, and the error distribution is in Fig. 110. The average error was 43mm, which is less than the hypothesized 50mm average error. The error never exceeded 100mm for points that occurred at least one second after abrupt gaze fixation changes. Therefore, as it was in Test 3, the second part of the hypothesis was also proven true.

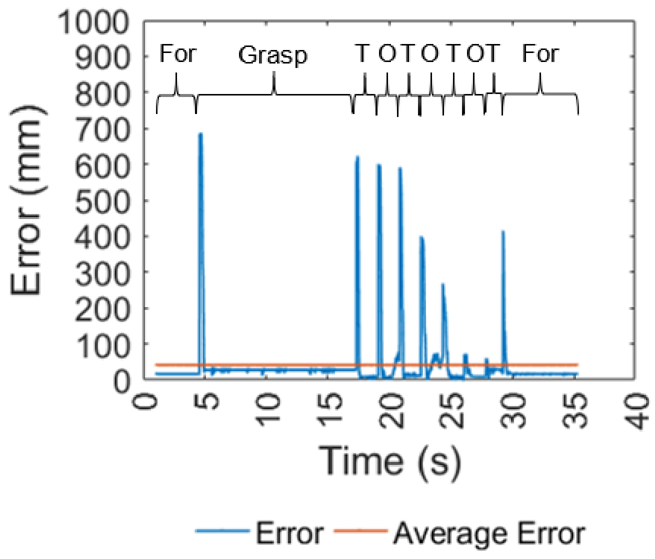


Fig. 109. Test 4: Magnitude of Error

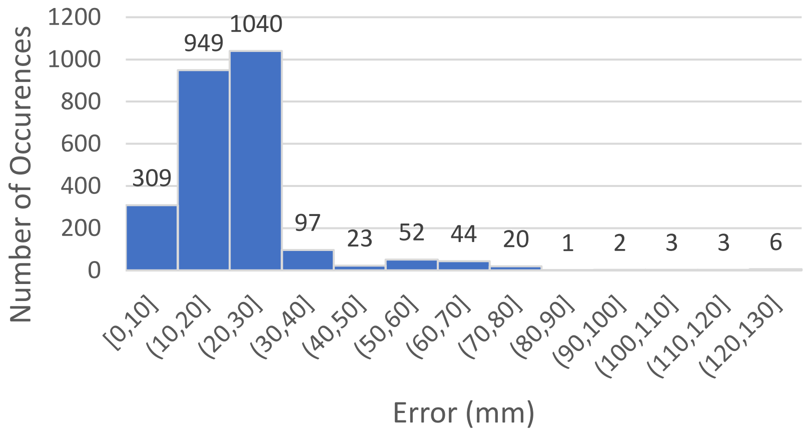


Fig. 110. Test 4: magnitude of error distribution

8.5. Linear Path: Tracking

The instructed and actual gaze fixation points during Test 5 are shown in Figures 111, 112 and 114. The gripper position began publishing to ROS about 16.6 seconds after the Arduino code was started, at which point the eyes began tracking the gripper.

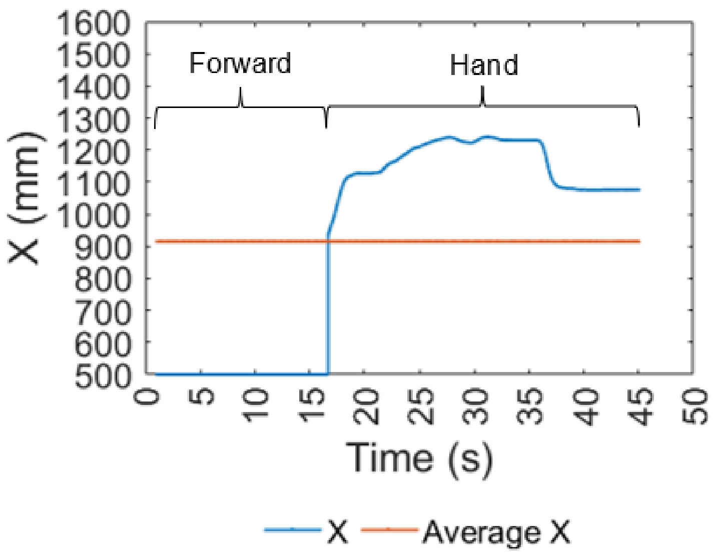


Fig. 111. Test 5: X vs. Time

Fig. 112 plots the y-values of the gripper location and actual gaze fixation point versus time. The average error was 20mm. The error distribution is in Fig. 113. The error fell within 48mm ninety percent of the time and within 55mm ninety-five percent of the time. As the gripper moved right to left, the eyes lagged behind the gripper in the y-direction by between half and one second, accounting for most of the error.

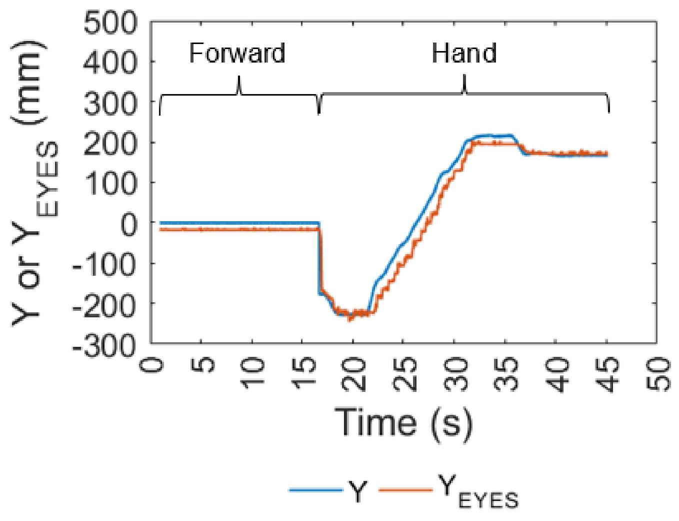


Fig. 112. Test 5: Lateral Motor Analysis

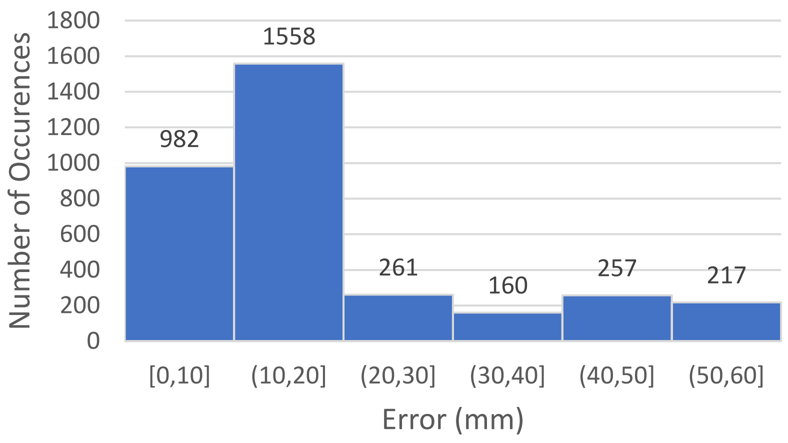


Fig. 113. Test 5: lateral motor error distribution

Fig. 114 plots the z-values of the gripper locations and actual gaze fixation points versus time. Again, the eyes tracked better in the z-direction than in the y-direction; the average error was 4.9mm compared to 20mm in y. The error distribution is in Fig. 115. The error fell within 13mm ninety percent of the time and within 19mm ninety-five percent of the time.

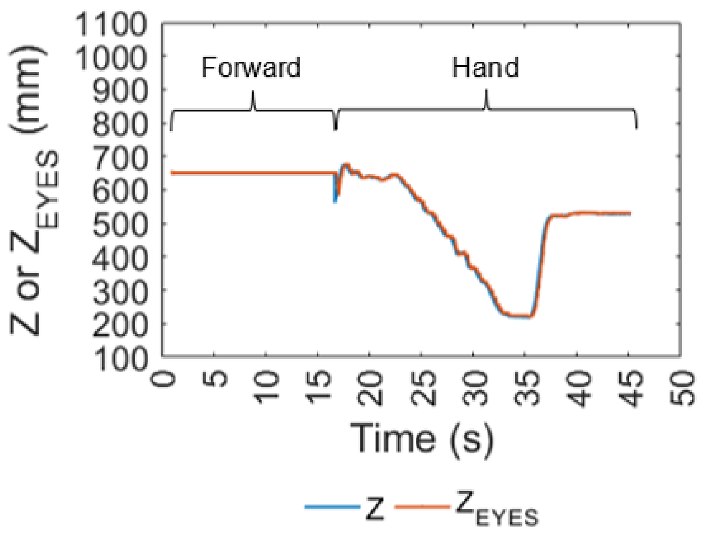


Fig. 114. Test 5: Vertical Motor Analysis

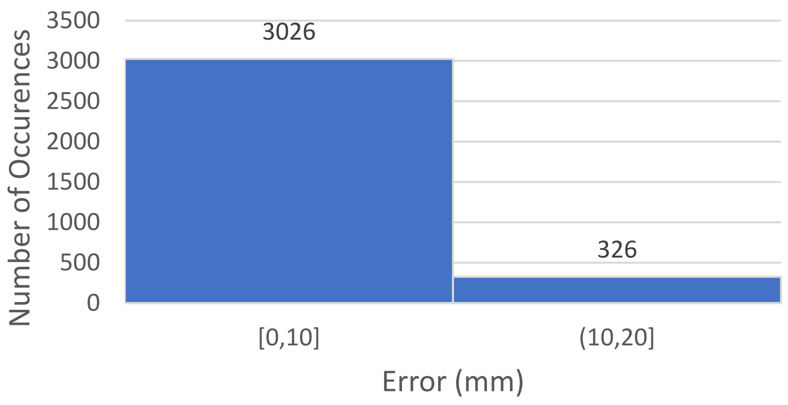


Fig. 115. Test 5: vertical motor error distribution

The magnitude of error is shown in Fig. 116. The average error was 22mm, which is less than the hypothesized 50mm. Excluding the spike in error when the eyes are first instructed to track the gripper, the largest magnitude of error was 77mm, below the hypothesized 100mm maximum. Both hypotheses were proven true.

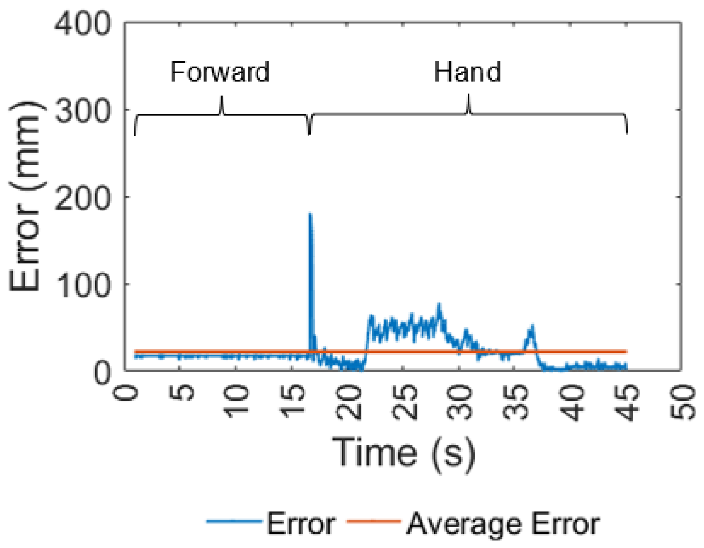


Fig. 116. Test 5: Magnitude of Error

The error distribution is in Fig. 117. The actual gaze fixation point was within 49mm of the instructed point ninety percent of the time and within 55mm ninety-five percent of the time.

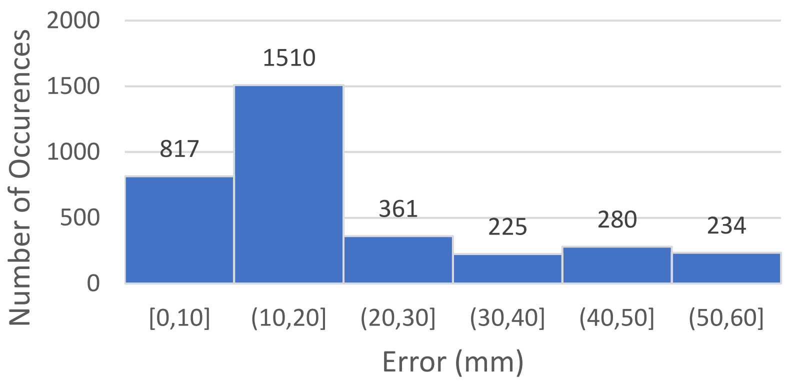


Fig. 117. Test 5: magnitude of error distribution

8.6. Linear Path: Block Mode

The instructed and actual gaze fixation points during Test 6 are shown in Figures 118, 119, and 121. The gripper position began publishing to ROS about 17.7 seconds after the Arduino code was started, at which point the eyes began looking at the grasp site.

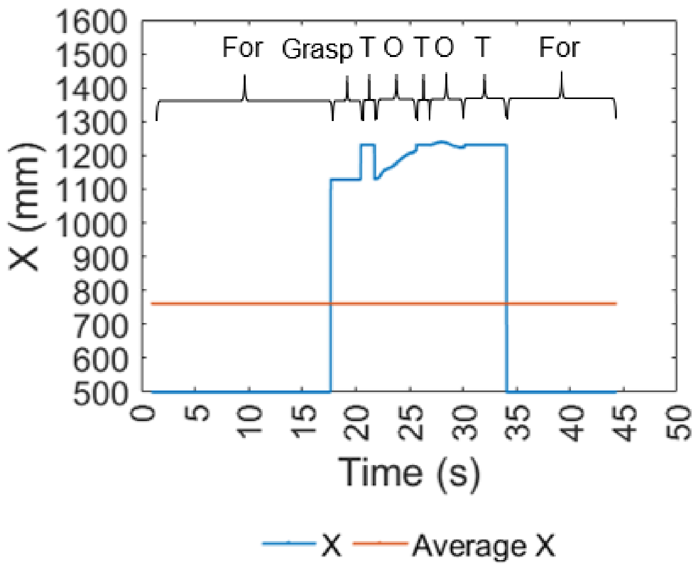


Fig. 118. Test 6: X vs. Time

Fig. 119 plots the y-values of the instructed and actual gaze fixation points versus time. The gripper's y location is also plotted from the first target fixation through the end of the last. The eyes always looked ahead of the gripper when instructed to during the offset phases. Again, there was some delay in achieving those instructed points.

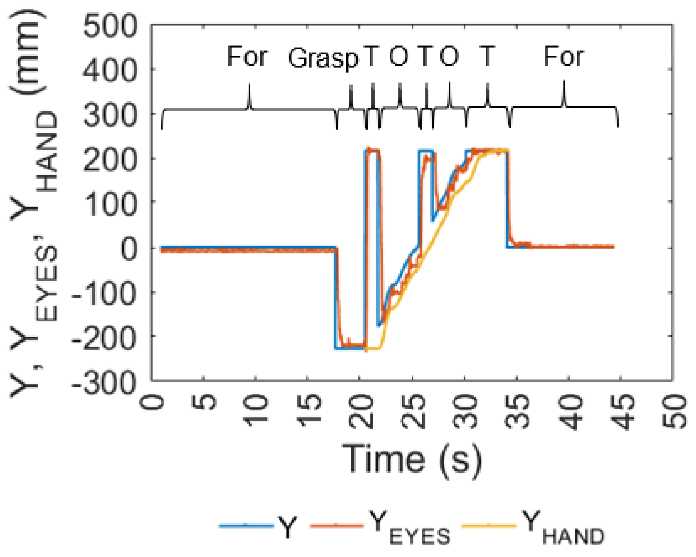


Fig. 119. Test 6: Lateral Motor Analysis

The error distribution is in Fig. 120. The average error was 18mm, and the error fell within 30mm ninety percent of the time and within 43mm ninety-five percent of the time.

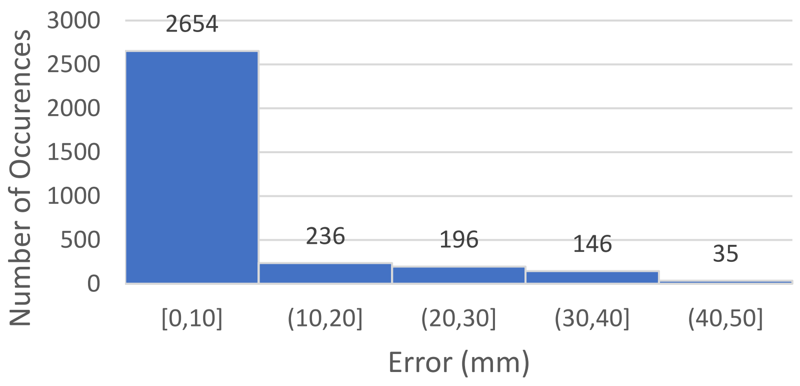


Fig. 120. Test 6: lateral motor error distribution

Fig. 121 plots the z-values of the instructed and actual gaze fixation points versus time. The gripper's z location is also plotted from the first target fixation through the end of the last. The average error was 14mm. Most larger errors occurred directly after abrupt changes in the instructed gaze fixation point. Additionally, for larger saccades such as those to and from the target, the vertical motor briefly overshot its instructed position. The behavior was the same as that observed with the parabolic path.

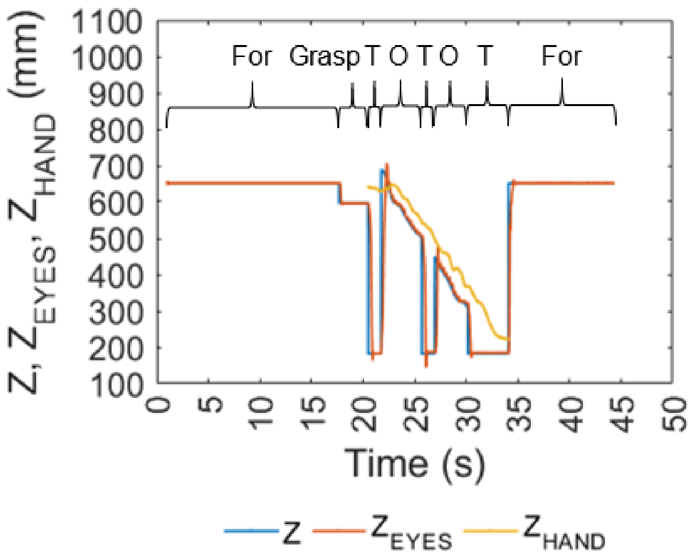


Fig. 121. Test 6: Vertical Motor Analysis

For approximately one second after the first target fixation, the eyes are instructed to look above the hand instead of below as desired. Recall that the trajectory was curve-fit; although the resulting equation fit the data extremely well (R^2=0.98), it did not reflect the more lateral hand motion at the start of the trajectory. An if statement could be written specifically for this trajectory that would prevent Z from exceeding Z_{HAND}.

The error distribution is in Fig. 122. The error fell within 9mm ninety percent of the time and within 45mm ninety-five percent of the time.

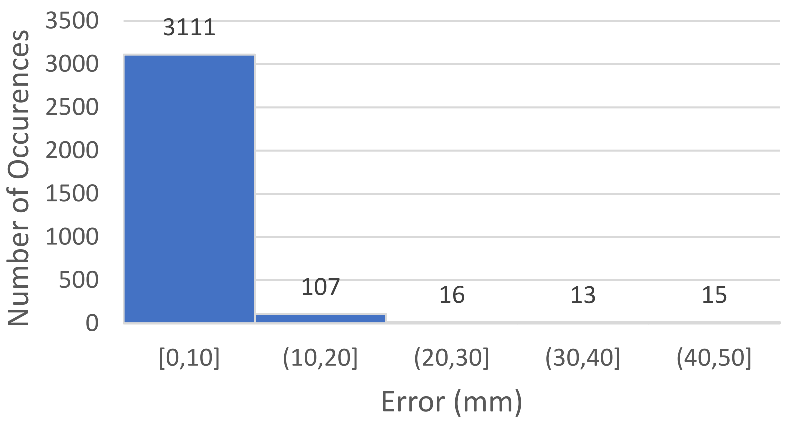


Fig. 122. Test 6: vertical motor error distribution

The magnitude of error is shown in Fig. 123. The average error was 25mm, which is within the hypothesized average error of 50mm. Apart from the spikes in error when gaze instructions changed abruptly, the error never exceeded 100mm for points that occurred at least one second after those spikes. Therefore, the second part of the hypothesis was also proven true.

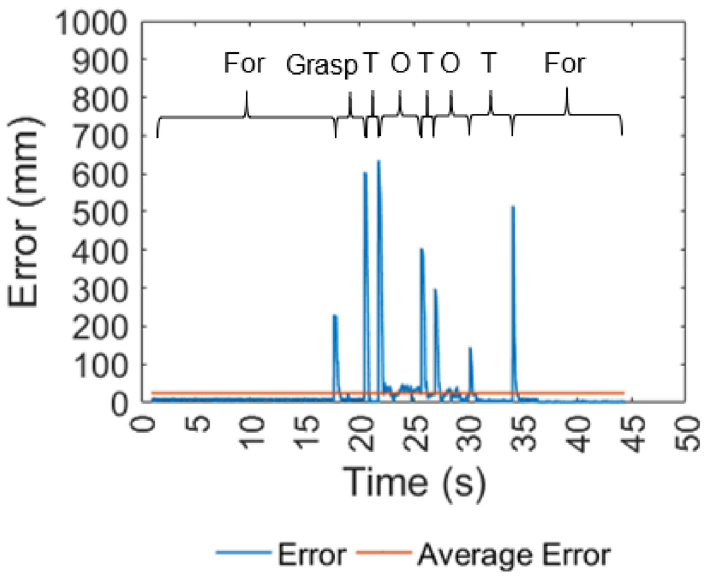


Fig. 123. Test 6: Magnitude of Error

The error distribution is in Fig. 124. The actual gaze fixation point was within 34mm of the instructed point ninety percent of the time and within 90mm ninety-five percent of the time.

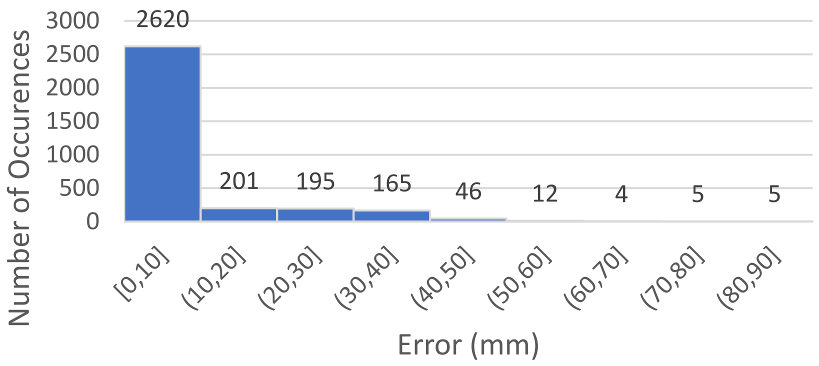


Fig. 124. Test 6: magnitude of error distribution

8.7. Linear Path: Full Glass Mode

The instructed and actual gaze fixation points during Test 7 are shown in Figures 125, 126, and 128. The gripper position began publishing to ROS 15.4 seconds after the Arduino code was started, at which point the eyes began looking at the grasp site.

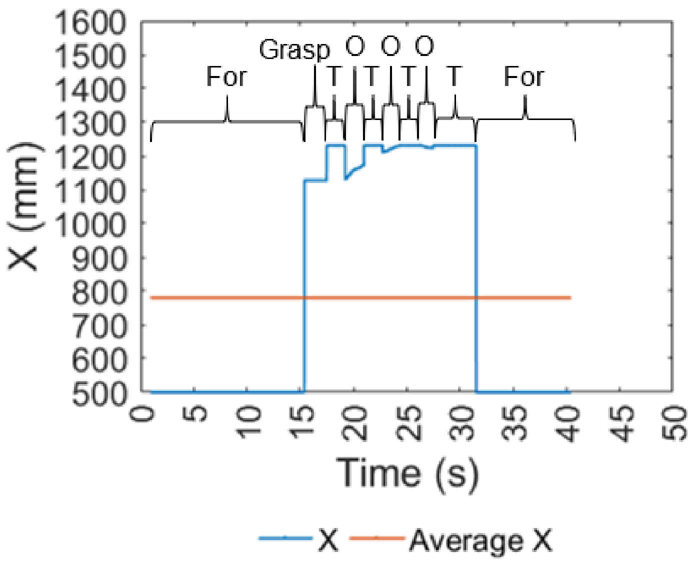


Fig. 125. Test 7: X vs. Time

Fig. 126 plots the y-values of the instructed and actual gaze fixation points versus time. The gripper's y location is also plotted from the first target fixation through the end of the last. The average error was 27mm. The steady-state error during target fixations was 20mm, equivalent 0.9° in this case. Unlike in the test running block mode, the eyes were not always ahead of the gripper; the delay in movement caused the eyes to look at or slightly ahead of the gripper during much of the offset phases. Those errors, however, were always under three degrees. The results are similar to those of the parabolic path.

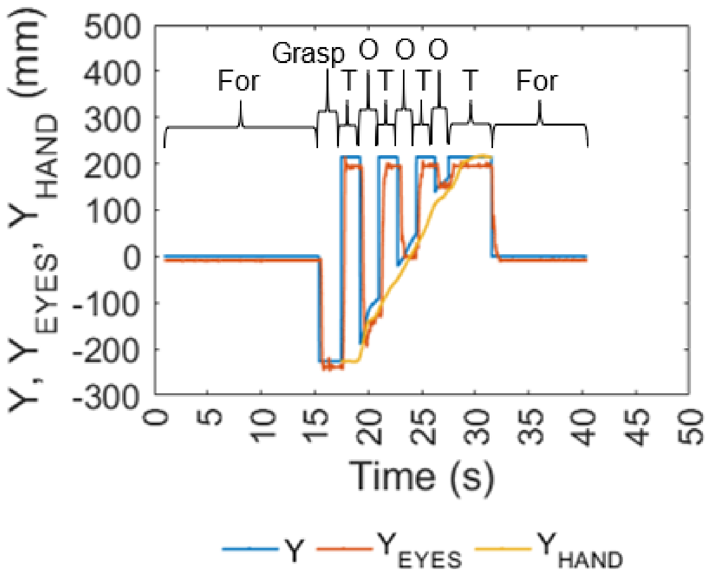


Fig. 126. Test 7: Lateral Motor Analysis

The error distribution is in Fig. 127. The error fell within 43mm ninety percent of the time and within 91mm ninety-five percent of the time.

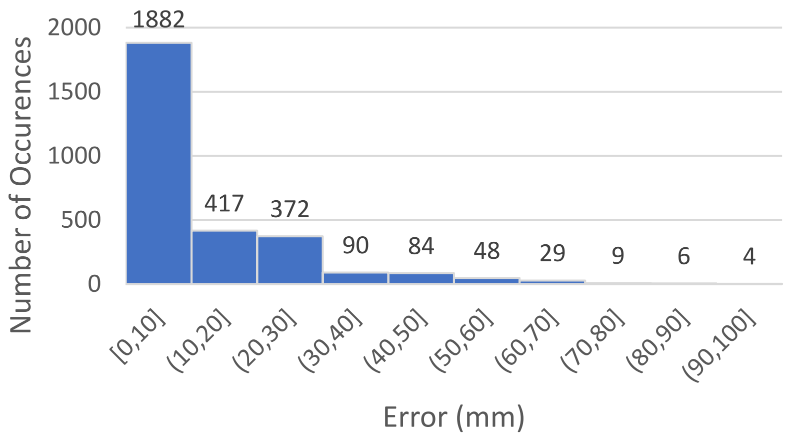


Fig. 127. Test 7: lateral motor error distribution

Fig. 128 plots the z-values of the instructed and actual gaze fixation points versus time. The gripper's y location is also plotted from the first target fixation through the end of the last. The average error was 20mm, only 7mm less than that of the lateral motor. Again, the vertical motor briefly overshot its instructed position during the larger saccades.

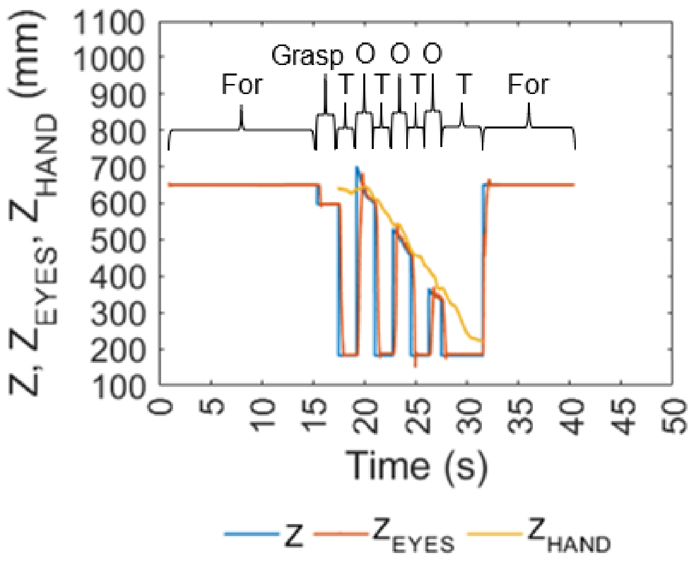


Fig. 128. Test 7: Vertical Motor Analysis

The error distribution is in Fig. 129. The error fell within 16mm ninety percent of the time and within 157mm ninety-five percent of the time.

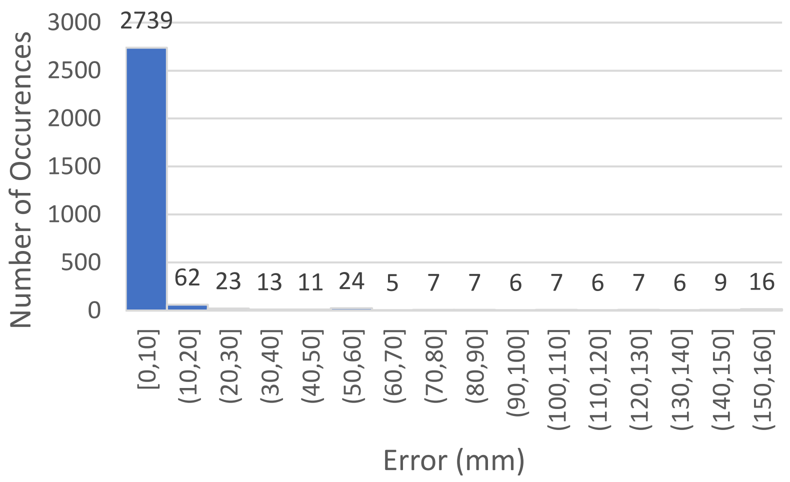


Fig. 129. Test 7: vertical motor error distribution

The magnitude of error is shown in Fig. 130, and the error distribution is in Fig. 131. The average error was 37mm, which is less than the hypothesized 50mm average error. The error never exceeded 100mm for points that occurred at least one second after abrupt gaze fixation changes. In all linear path tests, both hypotheses were proven true.

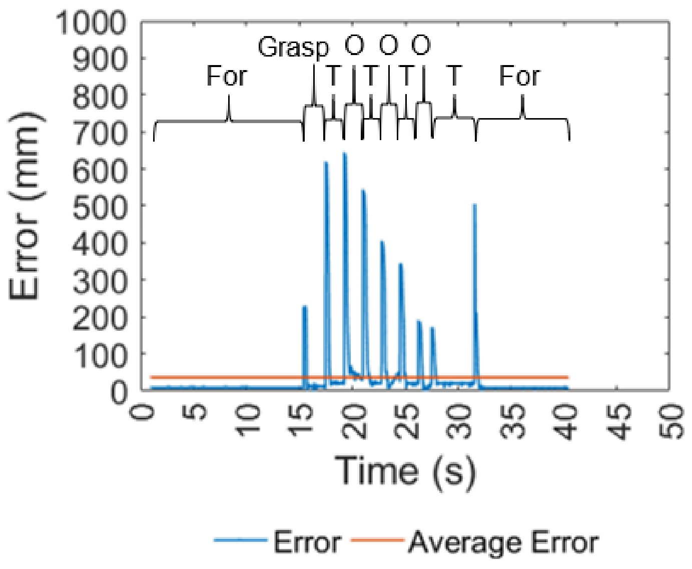


Fig. 130. Test 7: Magnitude of Error

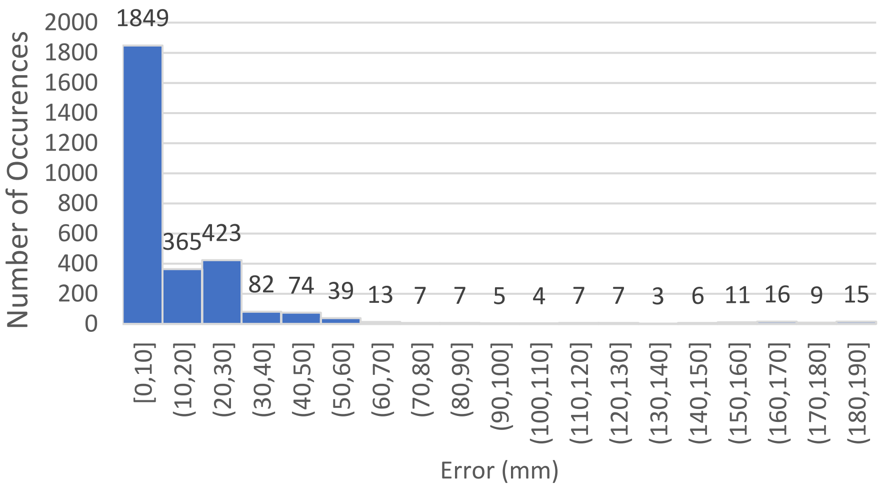


Fig. 131. Test 7: magnitude of error distribution

9 DISCUSSION

Replacing the 3D printed components with parts machined out of lightweight metal would create a more robust mechanism. The eyes' range of motion could also be increased through three adjustments. As stated in design section, the connection points that secure the two halves of each shell could be altered to increase horizontal range of motion. To increase vertical range of motion, one could use a longer threaded shaft or reduce the eyes' radius of rotation from 37mm to something smaller. Additionally, a new servo might fix the issue of the eyes tracking slightly worse right to left than vice versa. A fix to allow all eight Arduino publishers to run at once would reduce the work required to make a comprehensive analysis.

Further tests could be performed to verify the eyes' accuracy without relying on sensor output. The laser diode module eyes could be instructed to look at known coordinates marked on a solid surface parallel to the YZ plane. The midpoint between the two laser projections on the surface would be the actual gaze fixation point, and its distance from the instructed gaze fixation point would be the error. Sensor readings would be validated if the actual gaze fixation point calculated in the Arduino code matched that observed. Discrepancies would indicate all other sources of error not reflected in sensor output. One potential source would be mechanical imperfections such as those in linkages or from backlash in gears. Human error in measurement and the tolerance of the measuring tool would still influence the accuracy of external validation.

For applications in which the user would want to see what the robotic eyes see, eye design for the laser diode modules could be altered to house cameras instead. The two-piece eye design could also be modified to not reduce the assembly's range of motion as it does now.

Cosmetic changes that could be made to future design iterations include using paint to make the eyes and mask to look more lifelike. The addition of eyebrows, eyelashes, and eyelids would make the mask more realistic. Another option for an artificial face would be to mold one out of a silicone-based material like dragon skin. A flexible mask would be required regardless if additional points of actuation were added to the face, such as points at which to make the eyebrows or mouth move.

A task-based experiment using the robotic eyes, Baxter robot, and test subjects could be made to test if the eyes' programmed movement improves human-robot interaction. Improvement would be measured by the accuracy and efficiency of task completion.

10 CONCLUSIONS

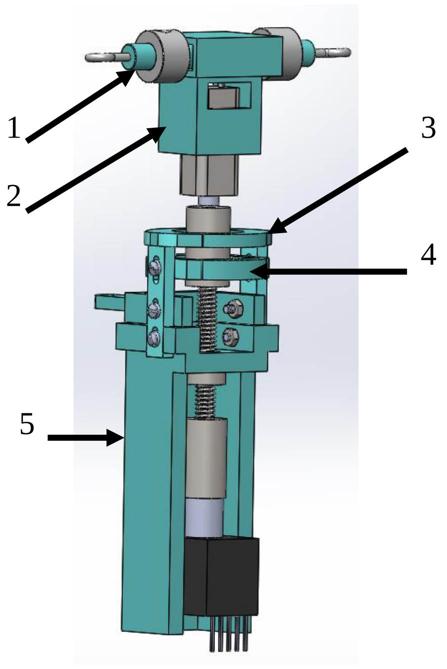
The overall goal to design and fabricate novel humanoid robotic eyes programmed to move logically with a Baxter robot based on hand-eye coordination research was achieved. Both position control for tracking and temporal control for human-like eye movement were achieved. Although the error in the lateral motor was greater than that of the vertical motor, both achieved average errors within 50mm throughout all tests with Baxter. The maximum error rarely exceeded the desired 100mm outside of abrupt changes in instructed gaze fixation points. Even the larger lateral motor errors only equated to less than four degrees while the average error was less than two degrees. Improvements could still be made to reduce the delay in obtaining instructed gaze fixation points. Lastly, temporal offsets should be made large enough in future application to ensure the eyes look ahead of the gripper when instructed to do so but small enough to remain human-like.

11 APPENDICES

Appendix A: Servo Calculations

The volumes of plastic components were found in SolidWorks. The component masses were found using SolidWorks Mass Properties unless otherwise stated.

VOLUME OF PLASTIC COMPONENTS	
COMPONENT	VOLUME (mm^3)
1	1653.19
2	5523.67
3	2412.90
4	1446.7
5	18517.03
TOTAL	**29553.49**



Density of PLA: 0.00125 grams/mm^3

The Cura printing infill was set at 20% for all plastic components.

$$\therefore m_{PLA\ components} = 0.2 * \rho_{PLA} * V_{PLA}$$

$$m_{PLA\ components} = 7.388\ grams = \boxed{0.07388kg}$$

$$l_{arm} = \frac{93}{64} inches * \frac{2.54cm}{1inch} = \boxed{3.69cm}$$

MASS OF LOAD COMPONENTS	
COMPONENT	MASS (GRAMS)
Coupler 1	11.56
Coupler 2	9.69
Threaded Rod	7
DC Motor (from datasheet)	17.69
Encoder (from datasheet)	11
Shaft Collars (combined mass)	3.96
Plastic Components	7.39
TOTAL	**68.29**

$$m_{total} = 68.29g = 6.829 * 10^{-2}kg$$

$$\boxed{\tau = g * m_{total} * l_{arm}}$$

$$\tau = 2.47N \cdot cm$$

Appendix B: Thread Pitch & DC Motor Calculations

In the following equations, x is the number of threads-per-inch, P is thread pitch, $\omega = 500°/s$ is the angular velocity of the eyeball, $r = 93/64"$ is the length of the moment arm, $V_{t,max}$ is the maximum tangential velocity, $V_{y,max}$ is the maximum velocity in the y-direction that was defined in Fig. 32 of the report, and $\omega_{DC,max}$ is the maximum angular velocity produced by the DC motor shaft in RPMs.

$$P = x^{-1}$$

$$\omega = 500\frac{°}{s} * \frac{1\,revolution}{360°} = 1.389\,rev/s$$

$$V_{t,max} = \omega r = \frac{1.389rev}{s} * \frac{93}{64}inches = 2.018\,inches/s$$

$$V_y = \omega r cos(\theta)$$

$$V_{y,max} = V_{t,max}$$

$$\omega_{DC,max} = \frac{V_{y,max}}{P} * \frac{60s}{1min}$$

$$\boxed{\omega_{DC,max} = \frac{121.08}{P}}$$

Note that the results in Table 1 of the report were produced in Excel. The above equation for $\omega_{DC,max}$ is not as accurate as the Excel calculations because the values of $V_{t,max}$ and ω above were rounded.

Appendix C: Circular Baxter Trajectory Code

```python
#!/usr/bin/env python


import rospkg
# rospy - ROS Python API
import rospy

# from std_msgs.msg import String, Float32, UInt8, Int32

# baxter_interface - Baxter Python API
import baxter_interface

import time

# initialize our ROS node, registering it with the Master (Master computer is on Baxter)
rospy.init_node('Hello_Baxter', anonymous=True)

# create an instance of baxter_interface's Limb class
limb = baxter_interface.Limb('left')

left_gripper = baxter_interface.Gripper('left')


pL1 = {'left_w0':-0.39231558650169457, 'left_w1':0.5821457090025145 , 'left_w2':0.0502378708032473 , 'left_e0':0.3911651009107805, 'left_e1':0.09088836168221076 , 'left_s0':-1.0852914074289302, 'left_s1':-0.8536603084582327}

pL2 = {'left_w0':-0.9717768291254096, 'left_w1': 0.9038981792614801, 'left_w2':0.06864564025787226 , 'left_e0':1.1401312205958338, 'left_e1': 0.14457768925820025, 'left_s0':-1.071869075534933, 'left_s1':-0.8816554578371415}

pL3 = {'left_w0':-0.24927187803137973, 'left_w1': 1.15892248524743, 'left_w2':-0.29529130166794215, 'left_e0':0.5437961893053792, 'left_e1': 0.24812139244046566, 'left_s0':-1.0032234352770606, 'left_s1':-0.9127185687918212}

pL6 = {'left_w0':-1.076471017898589, 'left_w1': 0.9779127522769513, 'left_w2':-0.07861651537912745, 'left_e0':1.194587538565766, 'left_e1': 0.23930100291012454 , 'left_s0':-1.424684656748578, 'left_s1':-0.5483981316690354}

pL5 = {'left_w0':-0.22281070944035636, 'left_w1': 1.148951610126175, 'left_w2':-0.30104372962251247, 'left_e0':0.3896311201228951, 'left_e1': 0.2891553785164005 , 'left_s0':-1.1746457883232555, 'left_s1':-0.7033301912454623}

pL4 = {'left_w0':-0.9805972186557508, 'left_w1': 1.19573802415668, 'left_w2':0.04065049087896346, 'left_e0':1.272820558747922, 'left_e1': 0.46096122675956686, 'left_s0':-1.2609322076418101, 'left_s1':-0.7152185423515741}

pL7 = {'left_w0':-0.565271920335775, 'left_w1': 0.6626797003664987, 'left_w2':-0.2891553785164005, 'left_e0':0.5832961945934286, 'left_e1': 0.09702428483375242 , 'left_s0':-1.3472186269603645, 'left_s1':-0.6254806662602774}

pL8 = {'left_w0':-0.7136845615636888, 'left_w1': 0.5951845456995405, 'left_w2':0.10929613113683573
, 'left_e0':0.7002622296696914, 'left_e1': 0.029529130166794215, 'left_s0':-
1.2570972556720965, 'left_s1':-0.7190534943212877}


left_gripper.calibrate()  # calibrate before opening or closing

left_gripper.open()  # open gripper before movement

#CW once (relative to observer)
limb.move_to_joint_positions(pL1)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL2)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL3)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL4)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL5)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL6)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL7)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL8)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL1)
time.sleep(1)  # in seconds

#CW again
limb.move_to_joint_positions(pL2)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL3)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL4)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL5)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL6)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL7)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL8)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL1)
time.sleep(1)  # in seconds

#CCW once
limb.move_to_joint_positions(pL8)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL7)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL6)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL5)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL4)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL3)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL2)
time.sleep(1)  # in seconds
```

```
limb.move_to_joint_positions(pL1)
time.sleep(1)  # in seconds

#CCW again
limb.move_to_joint_positions(pL8)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL7)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL6)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL5)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL4)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL3)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL2)
time.sleep(1)  # in seconds
limb.move_to_joint_positions(pL1)

quit()
```

Appendix D: Arduino Code and Pseudocode

Robotic_Eye_Code.ino

```
#include "dh_param.h"
#include "dc_motor_encoder_code.h"
#include "servo_pot_code.h"
#include "verification.h"

//This version has incorporated all hand-eye coordination code. The
target saccades are

// Pins being used (includes ones for motor driver not being utilized):
// 0,1,2,3,4,6,7,8,9,10.12,A0,A1,A2
// PWM disabled on 3 and 11

#include <ros.h> //located in ros_lib
#include <std_msgs/Float32.h>

unsigned long pTime1;
unsigned long pTime2;
unsigned long cTime2; //current time within sampling period
unsigned long o2Time; //assign once object = 2

int object = 1; //added

float mode = 3;  // 1 = Track Gripper
                 // 2 = "Block" Mode
                 // 3 = "Full Glass" Mode

float pos2R0Grip[3] = {500, 0, 650};   // gripper position initialized
same as pos2R0
float gap;                              // gap between gripper fingers,
from 0 to 100 as %
float graspSite[3] = {1128,-227,597};   // grasp site, {1188,557,382}
for parabolic path
                                        // {1128,-227,597} for linear
path

float target[3] = {1231,215,182};      // target, {1199,-75,350} for
parabolic path
                                       // {1231,215,182} for linear path

float tempY; //temporary y coordinate when using logic under object ==
2 condition

int targetInterval;        // eyes look at target every _ milliseconds
after grabbing object
int tOff;                  // time offset dependent on mode
int targetCondition;        // time remainder condition for looking at
target

float yVelAvg = 0.034;    // average y-velocity of hand moving from one
side to another, in mm/ms
                           // equals -0.051mm/ms for parabolic path
                           // 0.034mm/ms for linear path
```

```
float yOff;                  // y offset, product of tOff and yVelAvg

ros::NodeHandle nh;

void messageCbx(const std_msgs::Float32& gripper_msg_x){ //name of
message is gripper_msg
  pos2R0Grip[0] = gripper_msg_x.data * 1000;  //gripper x, convert from
meters to mm

  if (mode == 1)
  {
    parameterClass.pos2R0[0]= pos2R0Grip[0]; //follow gripper in mode 1
  }
  else if (object == 1) // haven't grabbed object
  {
    parameterClass.pos2R0[0]=graspSite[0]; //look at grasp site until
right before item is grabbed
  }
  else if (object == 2) // almost grabbed or have object
  {
    if ((millis()-o2Time) % targetInterval <= targetCondition ||
millis()-o2Time >= 10000)
    {
      parameterClass.pos2R0[0]= target[0];  //look at target
    }
    else
    {
      parameterClass.pos2R0[0]= pos2R0Grip[0]; // xEyes same as xHand
when holding object
    }
  }
  else if (object == 3) // letting go of or releasing object
  {
    parameterClass.pos2R0[0]= 500; //look straight ahead when object
released
  }
  else
  {
    //nothing, shouldn't reach this
  }

}

void messageCby(const std_msgs::Float32& gripper_msg_y){ //name of
message is gripper_msg
  pos2R0Grip[1] = gripper_msg_y.data * 1000;  //gripper y, convert from
meters to mm

  if (mode == 1)
  {
    parameterClass.pos2R0[1]= pos2R0Grip[1];
  }

  else if (object == 1)
  {
    parameterClass.pos2R0[1]=graspSite[1]; //look at grasp site until
right before item is grabbed
  }

  else if (object == 2)
  {

   if ((millis()-o2Time) % targetInterval <= targetCondition ||
millis()-o2Time >= 10000) //2nd time condition for linear path only
   {
     tempY = target[1];
   }
   else //look at offset
   {
     tempY = pos2R0Grip[1] + yOff;
   }

   //Assing value to pos2R0[1]
   if (tempY > target[1]) //< sign for parabolic path (L -> R), > sign
for linear path (R -> L)
   {
    parameterClass.pos2R0[1] = target[1]; // prevent eyes from looking
at y past target
   }
   else
   {
    parameterClass.pos2R0[1] = tempY;
   }
  }

  else if (object == 3)
  {
    parameterClass.pos2R0[1]= 0; //look straight ahead when object
released
  }

  else
  {
    //nothing, shouldn't reach this
  }
}

void messageCbz(const std_msgs::Float32& gripper_msg_z){ //name of
message is gripper_msg
  pos2R0Grip[2] = gripper_msg_z.data * 1000;  //gripper z, convert from
meters to mm

  if (mode == 1)
  {
    parameterClass.pos2R0[2]= pos2R0Grip[2];
  }
  else if (object == 1)
  {
    parameterClass.pos2R0[2]= graspSite[2]; //look at grasp site until
right before item is grabbed
  }
  else if (object == 2)
  {
   if ((millis()-o2Time) % targetInterval <= targetCondition ||
millis()-o2Time >= 10000) //2nd time condition for linear path only
   {
     parameterClass.pos2R0[2]= target[2];
   }
   else //look at offset
   {
//     parameterClass.pos2R0[2]= -0.0015*sq(parameterClass.pos2R0[1]) +
0.6938*parameterClass.pos2R0[1] + 462.28; //calculate using instructed
gaze y-coor. and curve-fit parabolic path
     parameterClass.pos2R0[2]= -1.0226*parameterClass.pos2R0[1] +
506.94; //linear path
   }
  }
  else if (object == 3)
  {
    parameterClass.pos2R0[2]=650; //look straight ahead when object
released, more consistent than gripper coordinate condition
  }
  else
  {
    //nothing, shouldn't reach this
  }
}

void messageCbGp(const std_msgs::Float32& gripper_msg_gp){ //name of
message is gripper_msg
  gap = gripper_msg_gp.data;  //percent gripper is open

  if (object == 1)
  {
    if (gap < 30)
    {
      object = 2; //doesn't keep assigning object = 2
      o2Time = millis(); //time when object first = 2, only assigned
once
    }
  }
  else if (object == 2)
  {
    if (gap >= 30)
    {
      object = 3; //doesn't confuse haven't gotten object and released
object since assigned after object = 2
    }
  }
  else
  {
    //nothing, shouldn't reach this
  }
}

std_msgs::Float32 timeStamp; //added
std_msgs::Float32 testx;
std_msgs::Float32 testy;
std_msgs::Float32 testz;
```

```
std_msgs::Float32 testya;
std_msgs::Float32 testza;
//std_msgs::Float32 testyg;
//std_msgs::Float32 testzg;

ros::Subscriber<std_msgs::Float32> suba("Baxter_x", &messageCbx);
//subscribes to topic
ros::Subscriber<std_msgs::Float32> subb("Baxter_y", &messageCby);
//subscribes to topic
ros::Subscriber<std_msgs::Float32> subc("Baxter_z", &messageCbz);
//subscribes to topic
ros::Subscriber<std_msgs::Float32> subd("Gripper_Pos", &messageCbGp);
//subscribes to topic

ros::Publisher chatterTime("chatterTime", &timeStamp); //added
ros::Publisher chatterx("chatterx", &testx);
ros::Publisher chattery("chattery", &testy);
ros::Publisher chatterz("chatterz", &testz);
ros::Publisher chatterya("chatterya", &testya);
ros::Publisher chatterza("chatterza", &testza);
//ros::Publisher chatteryg("chatteryg", &testyg);
//ros::Publisher chatterzg("chatterzg", &testzg);

void setup(){

  // Mode conditions
  if (mode == 2)
  {
    targetInterval = 5380;
    tOff = 1500;
    targetCondition = 1300;
  }
  else if (mode == 3)
  {
   targetInterval = 3510;
   tOff = 1000;
   targetCondition = 1755;
  }
  yOff = yVelAvg * tOff;  //calculate once after assigning tOff
  servoClass.SETUP1();    //attach servo
  servoClass.SETUP2();    //defines servo initial position as an angle
  motorClass.SETUP(); //pinmode, attaches interrupts, defines shield as
md


  //publisher & subscriber code
  nh.initNode(); //initialize ros node handle

  nh.advertise(chatterTime); //added
  nh.advertise(chatterx); //advertise to topics for input versus actual
points
  nh.advertise(chattery);
  nh.advertise(chatterz);
  nh.advertise(chatterya);
  nh.advertise(chatterza);
//  nh.advertise(chatteryg);
//  nh.advertise(chatterzg);
```

```
  nh.subscribe(suba); //subscribe to topic
  nh.subscribe(subb);
  nh.subscribe(subc);
  nh.subscribe(subd); //added for gripper gap


  verificationClass.homeSetup(); //defines initial position of bar
centroid relative to eye midpoint
  parameterClass.SETUP(); //calculates initial deltas and other
coordinates (deltas all zero)
  pTime1 = millis();

}

void loop(){

  if (cTime2>=10) //sampling at 100Hz -> period of 10ms
  {
    //Publish input versus actual calculated in verification tab to
chatter topic
    timeStamp.data = (millis());
    testx.data = parameterClass.pos2R0[0];
    testy.data = parameterClass.pos2R0[1];
    testz.data = parameterClass.pos2R0[2];
    testya.data = verificationClass.aPos2R0[1];
    testza.data = verificationClass.aPos2R0[2];
//    testyg.data = pos2R0Grip[1];
//    testzg.data = pos2R0Grip[2];

    chatterTime.publish ( &timeStamp );
    chatterx.publish( &testx );
    chattery.publish( &testy );
    chatterz.publish( &testz );
    chatterya.publish( &testya );
    chatterza.publish( &testza );
//    chatteryg.publish( &testyg );
//    chatterzg.publish( &testzg );
    nh.spinOnce();
    //don't need delay, already runs at servoTime interval

    verificationClass.SETUP(); //find actual point and end of run

    pTime2=millis();
  }

  if (servoClass.cTime1 >= servoClass.servoTime && servoClass.cTime1 %
servoClass.servoTime <= 5)
  {
    verificationClass.SETUP(); //find actual point and end of run
    parameterClass.SETUP(); //find deltas
    servoClass.SETUP2(); // reset initial servo position
  }

  servoClass.cTime1 = millis() - pTime1; //update servoClass.cTime1
  servoClass.servoLoop(); // move servo
  motorClass.motorLoop(); // move motor
  if (servoClass.cTime1 >= 300) //Timer 1 dual purpose, used to start
new instruction loop and then reset for feedback loop
                                                                  //time
condition is servoTime PLUS time allotted for correction (remember same
timer)
  {
    pTime1=millis(); // restarts Timer 1
    motorClass.previousStart = millis(); //redefines after each
motorTime cycle passes
  }

  cTime2 = millis() - pTime2; //update Timer2
}
```

dc_motor_encoder_code.cpp

```
#include "Arduino.h"
#include "dc_motor_encoder_code.h"
#include "dh_param.h"

#include "DualMC33926MotorShield.h"

DualMC33926MotorShield md;

#define encoder0PinA  2
#define encoder0PinB  3
static volatile int encoder0Pos; // was volatile unsigned int

mClass::mClass(){}

void mClass::SETUP(){

  pinMode (encoder0PinA, INPUT);
  pinMode (encoder0PinB, INPUT);

  // encoder pin on interrupt 0 (pin 2)
  attachInterrupt(0, doEncoderA, CHANGE);

  // encoder pin on interrupt 1 (pin 3)
  attachInterrupt(1, doEncoderB, CHANGE);

  md.init(); //forgot before

  motorClass.previousStart = millis();
}

void mClass::stopIfFault()
{
  if (md.getFault())
  {
    Serial.println("fault");
    while(1);
  }
}

void mClass::doEncoderA() {
  // look for a low-to-high on channel A
  if (digitalRead(encoder0PinA) == HIGH) {

    // check channel B to see which way encoder is turning
    if (digitalRead(encoder0PinB) == LOW) {
      encoder0Pos = encoder0Pos + 1;         // CW
    }
    else {
      encoder0Pos = encoder0Pos - 1;         // CCW
    }
  }

  else   // must be a high-to-low edge on channel A
  {
    // check channel B to see which way encoder is turning
    if (digitalRead(encoder0PinB) == HIGH) {
      encoder0Pos = encoder0Pos + 1;          // CW
    }
    else {
      encoder0Pos = encoder0Pos - 1;          // CCW
    }
  }

}

void mClass::doEncoderB() {
  // look for a low-to-high on channel B
  if (digitalRead(encoder0PinB) == HIGH) {

    // check channel A to see which way encoder is turning
    if (digitalRead(encoder0PinA) == HIGH) {
      encoder0Pos = encoder0Pos + 1;         // CW
    }
    else {
      encoder0Pos = encoder0Pos - 1;         // CCW
    }
  }

  // Look for a high-to-low on channel B

  else {
    // check channel B to see which way encoder is turning
    if (digitalRead(encoder0PinA) == LOW) {
      encoder0Pos = encoder0Pos + 1;          // CW
    }
    else {
      encoder0Pos = encoder0Pos - 1;          // CCW
    }
  }
}

void mClass::motorControl(){

currentMillisM = millis();  //redundant for ramp conditions

  stopIfFault(); //stops motor if problem

   //RAMP UP
    if (currentMillisM - previousStart < rampTime) //make sure in
testing that i=400 is reached in this time
    {
      if (RPM <= maxRPM)  //was maxSpeed
      {
        RPM = RPM + delayMotorMultiplier; //more positive
        i=round(-0.000004 * sq(RPM) + 0.0912 * RPM + 55.364);
        if (parameterClass.deltaZ >= 0)
        {
          md.setM1Speed(i);  // CW if positive parameterClass.deltaZ,
may need to switch cases
         }
        else
        {
          md.setM1Speed(-1*i);  //CCW if negative parameterClass.deltaZ
        }
      }
    }

    //RAMP DOWN
    else if (currentMillisM - previousStart >= 0.8 * motorTime)
    {
      if (RPM >= 0) // could change to 5 to keep motor moving if needs
more time
      {
        RPM = RPM - delayMotorMultiplier;  //was plus (less negative)
now minus
        i=round(-0.000004 * sq(RPM) + 0.0912 * RPM + 55.364);
        if (parameterClass.deltaZ < 0)
        {
          md.setM1Speed(-1*i);  //CCW if negative parameterClass.deltaZ
        }
        else
        {
          md.setM1Speed(i);  // CW if positive parameterClass.deltaZ,
may need to switch cases
        }
      }
    }

    else
    {
      if ((currentMillisM - previousStart - rampTime) < 10) //reach max
rpm if not yet reached
      {
        rampPos = encoder0Pos *-1 * 25.4 / (400 * 24);
        totalPos = parameterClass.pos3R1[2]; //desired position for end
of motorTime
        bufferRPM = abs((60000 / 25.4) * (totalPos - rampPos) / (0.7 *
motorTime)); // convert mm/ms to RPM
        RPM = maxRPM + bufferRPM;
        i=round(-0.000004 * sq(RPM) + 0.0912 * RPM + 55.364);

        if (parameterClass.deltaZ>0)
        {
          md.setM1Speed(i);
        }
        else
        {
          md.setM1Speed(-1*i);  // stop motor if number of rotations
reached early
        }

      }
      else
      {
      }
    }
}
```

```
void mClass::motorCorrection(){
 if (abs(encoder0Pos * -1 * 25.4 / (400 * 24) -
parameterClass.pos3R1[2]) > 0.1) //if MOTOR is over 1mm from desired
position
  {
    if ((encoder0Pos * -1 * 25.4 / (400 * 24)) -
parameterClass.pos3R1[2] >= 0) //motor higher than desired position
    {
      md.setM1Speed(-125);
    }
    else //motor lower than desired position
    {
      md.setM1Speed(125);
    }
  }
  else //put stop motor in here instead if motor within acceptable
error
  {
    RPM=0;
    i=round(-0.000004 * sq(RPM) + 0.0912 * RPM + 55.364);
    md.setM1Speed(i);  // stop motor if number of rotations reached
  }
}

void mClass::motorLoop(){
  stopIfFault();
  doEncoderA();
  doEncoderB();
  eTicks = encoder0Pos;

  if (servoClass.cTime1 <= motorTime)
  {
    motorRotations = ceil(abs(parameterClass.deltaZ) * 24 *
(1/25.4));  // 24 rotations per inch displacement, in to mm
    rampTime = 0.2 * motorTime;  //40% of time used for ramp up/down,
in milliseconds
  //  avgMotorRPM = 60000 * motorRotations / motorTime;  //convert
milliseconds to minutes, in RPM

    maxRPM = (60000 * abs(parameterClass.deltaZ) * 24 * (1/25.4) /
motorTime) / 0.8;   //average value theorem dependent on ramp time
    maxRPM = constrain(maxRPM, 0, 4901); // ADDED, cannot exceed spec'd
maximum +1

    //int maxSpeed = map(maxRPM, 0, 4900, 0, 400);  //CHANGEd TO
EQUATION FOUND IN EXCEL
    maxSpeed = round(-0.000004 * sq(maxRPM) + 0.0912 * maxRPM +
55.364);

    // delayMotor = rampTime/maxRPM;  // ramp up time divided by number
of increments of RPM, was i
    delayMotorMultiplier = ceil(4 *maxRPM / rampTime); //dependent on
time increment chosen in loop

    currentMillisM = millis();  //redundant for if statement
    if (currentMillisM - previousMillisM >= 4)  // was >= delayMotor
    {
      previousMillisM = currentMillisM;
      motorControl();
    }
  }
  else
  {
    motorCorrection();
  }
}

mClass motorClass = mClass();
```

dc_motor_encoder_code.h

```
#ifndef dc_motor_encoder_code_h
#define dc_motor_encoder_code_h

#include "servo_pot_code.h"
//Pins used by the motor shield: 4,7,8,9,10.12,A0,A1

class mClass
{
  public:
    mClass();
    static void SETUP();
    void stopIfFault();
    static void doEncoderA();
    static void doEncoderB();
    int eTicks; //added, couldn't have encoder0Pos here, need to use in
verification.cpp
    void motorControl();
    void motorLoop();

    void motorCorrection(); //ADDED to correct motor position

    int i; // increment for motor speed, was zero
    int RPM; //added, current RPM

    int motorTime = 115; // time allotted for motor to run in milliseconds,
linked to servoTime

    int motorRotations;
    int rampTime;
    int maxRPM;
    int maxSpeed;
    int delayMotorMultiplier; // likely should give better name

    float rampPos;
    float totalPos;
    int bufferRPM;

    unsigned long currentMillisM=millis();
    unsigned long previousMillisM;        // will store last time speed was
updated
    unsigned long previousStart;


};

extern mClass motorClass;

#endif
```

dh_param.cpp

```
#include "dh_param.h"
#include "math.h"
#include "Arduino.h"
#include "verification.h"

dhClass::dhClass(){}

void dhClass::SETUP() {
 //Gripper wrt eye midpoint = gripper wrt base minus midpoint wrt base
  pos2R1[0] = pos2R0[0]-pos1R0[0];
  pos2R1[1] = pos2R0[1]-pos1R0[1];
  pos2R1[2] = pos2R0[2]-pos1R0[2];

  magA= sqrt(sq(pos2R1[0]) + sq(pos2R1[1]) + sq(pos2R1[2]));
//magnitude of midpoint to gripper vector, square root of sum each
pos2R1 element squared


  //Find bar centroid wrt base
  pos3R0[0] = pos1R0[0] - ((radius * pos2R1[0]) / magA);
  pos3R0[1] = pos1R0[1] - ((radius * pos2R1[1]) / magA);
  pos3R0[2] = pos1R0[2] - ((radius * pos2R1[2]) / magA);

// define previous centroid based on actual position from previous run
  // cant equate array to array, must go do so element-wise
  for (int entry=0; entry<=2; entry++)
  {
    oldPos3R1[entry] = verificationClass.aPos3R1[entry];
  }

  //Find desired bar centroid relative to fixed midpoint
  //Equals bar centroid wrt base minus midpoint wrt base
  pos3R1[0]=pos3R0[0]-pos1R0[0];
  pos3R1[1]=pos3R0[1]-pos1R0[1];
  pos3R1[2]=pos3R0[2]-pos1R0[2];

  //Find delta x, y, z
  deltaX=pos3R1[0]-oldPos3R1[0];
  deltaY=pos3R1[1]-oldPos3R1[1];
  deltaZ=pos3R1[2]-oldPos3R1[2];

}

dhClass parameterClass = dhClass();
```

dh_param.h

```
#ifndef dh_param_h
#define dh_param_h

#include <math.h>
class dhClass
{
  public:
    dhClass();
    void SETUP();
    //int pos2R0[];   //gripper relative to base from .csv file
    float pos2R1[3];  //midpoint to desired gripper trajectory point
    float magA; //magnitude of vector from midpoint to desired gripper point

    float pos3R0[3]; //bar centroid relative to base
    int radius = 37; //distance from eye midpoint to bar centroid in
millimeters

    float pos3R1[3] = {-37, 0, 0};  //give initial position of bar centroid
relative to fixed eye midpoint,

    float oldPos3R1[3]; ////previous bar centroid position

    float deltaX;
    float deltaY;
    float deltaZ;

    //Position Vectors
    //*Put from here down in loop*
    float pos1R0[3] = {0,0,650}; //fixed eye midpoint relative to Baxter base
measured, was set to approx 900

    float pos2R0[3] = {500, 0, 650};  //initialize to look straight ahead at
first before point obtained from Baxter
};

extern dhClass parameterClass;

#endif
```

servo_pot_code.cpp

```
#include "Arduino.h"
#include "servo_pot_code.h"
#include "dh_param.h"

ServoTimer2 myservo;

sClass::sClass(){

}

// run in robotic setup ONLY
void sClass::SETUP1(){
  myservo.attach(servoPin);

}

// redefine initial position with every new gripper point
void sClass::SETUP2(){
  currentPosition=getPos(feedbackPin); // degrees
  initialPosition=currentPosition;  //degrees
}

// Move servo to position
void sClass::Seek(ServoTimer2 servo, int analogPin, float pos)
{

  // Writes defined angle to motor in increments (start moving)
  if (abs(getPos(feedbackPin) - pos) >= 1)  //if position read by pot
is not within _ degrees of instructed position
  {
    if (getPos(feedbackPin) < pos)
    {
      currentPosition++;
      currentPosition = constrain(currentPosition, 62, angle);
    }
    else
    {
      currentPosition--;
      currentPosition = constrain(currentPosition, angle, 132);
    }

    servo.write(1800 - (((128 - currentPosition) * (1800 - 1300)) /
(128 - 80)));
  }
  else // if off by less than a degree, go straight to desired angle
  {
    servo.write(1800 - (((128 - angle) * (1800 - 1300)) / (128 - 80)));
  }
}


void sClass::Seek2(ServoTimer2 servo)
{
  servo.write(1800 - (((128 - angle) * (1800 - 1300)) / (128 - 80)));
}


float sClass::getPos(int analogPin)
{
 return (maxDegrees - (((maxFeedback - analogRead(analogPin)) *
(maxDegrees - minDegrees)) / (maxFeedback - minFeedback)));
}


void sClass::servoLoop(){
  currentMillisS = millis();
  deltaAngle = (acos(parameterClass.pos3R1[1] / (sqrt(sq(radius) -
sq(parameterClass.pos3R1[2])))) - ((187 - initialPosition) * M_PI /
180)) * -1 * 180 / M_PI;
  delayServo =  round(abs(servoTime/deltaAngle));
  if (delayServo < 2) //servo delay cannot feasibly be less than 2
milliseconds
  {
    delayServo = 2; // ignore calculation if less than 2 and assign 2
  }
  if (cTime1 >= servoTime + 50)
  {
    currentMillisS=millis();
    if (currentMillisS - previousMillisS >= delayServo)
    {
      Seek2(myservo);  // moves to instructed angle
      previousMillisS = currentMillisS;
    }
  }
  else if (currentMillisS - previousMillisS >= delayServo)
  {
    angle = initialPosition + deltaAngle;
    angle = constrain(angle, 62, 132); //ADDED so servo doesn't try to
go beyond feasible range
    Seek(myservo, feedbackPin, angle);  // moves to __ degree mark
    previousMillisS = currentMillisS;
  }
}




sClass servoClass = sClass();
```

servo_pot_code.h

```
#ifndef servo_pot_code_h
#define servo_pot_code_h

// ServoTimer2 library disables analogWrite PWM on pins 3 & 11 (vs 9 & 10 on
regular servo library)
// Can still use pins 3 and 11 for other purposes
#include <ServoTimer2.h>
#include <math.h>
#include "Arduino.h"


class sClass
{
  public:
    sClass();
    void SETUP1();
    void SETUP2();
    void Seek(ServoTimer2 servo, int analogPin, float pos);
    float getPos(int analogPin);
    void servoLoop();

    unsigned long cTime1; //current time within a motor run

    void sClass::Seek2(ServoTimer2 servo);

    // Control and feedback pins
    #define servoPin 5
    int feedbackPin = A5;  //was A5

    int radius = 37; // radius in mm, already in DH parameter code

    // Calibration values
    int minDegrees=62;
    int maxDegrees=132;
    int minFeedback=725;  // was 965, 713
    int maxFeedback=962;  // was 735, 951

    int servoTime = 100;  // in milliseconds, time allotted for servo movement

    float deltaAngle; //change in servo angle
    int delayServo;  //delay allowed for servo
    float angle;  //desired final angle of servo (pos argument of void Seek())

    float initialPosition; // angle in deg from pot at start
    float currentPosition; // angle in deg from pot

    unsigned long previousMillisS = 0;        // will store last time speed
was updated, was zero
    unsigned long currentMillisS=millis();
};
```

```
extern sClass servoClass;

#endif
```

verification.cpp

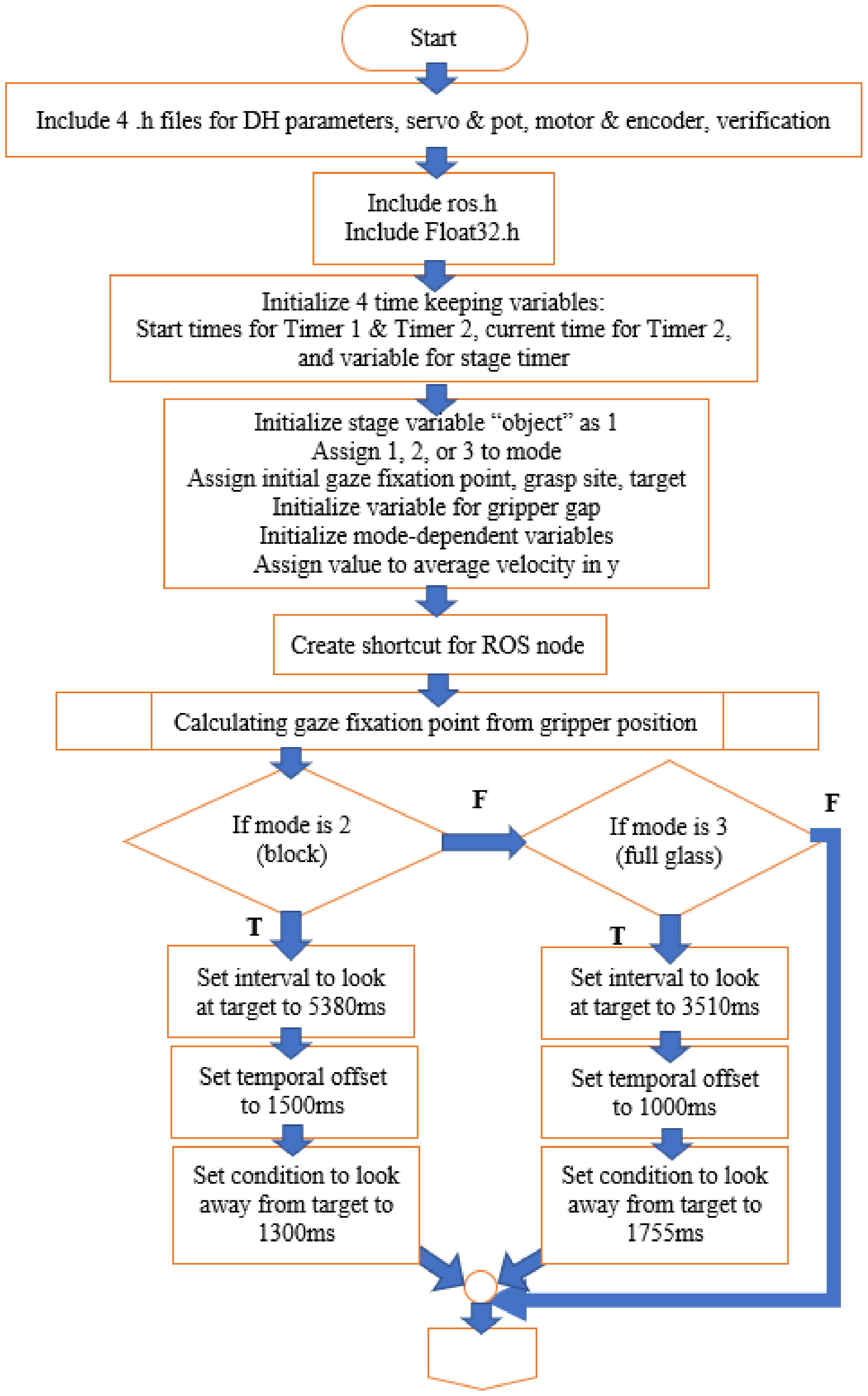
```
#include "verification.h"
#include "math.h"
#include "Arduino.h"

#include "servo_pot_code.h"
#include "dc_motor_encoder_code.h"
#include "dh_param.h"

vClass::vClass(){}

void vClass::homeSetup(){
  aPos3R1[0] = -37;
  aPos3R1[1] = 0;
  aPos3R1[2] = 0;
}

void vClass::SETUP() {
 // (A) FIND ACTUAL BAR CENTROID RELATIVE TO EYE MIDPOINT

 finalPosition = servoClass.getPos(servoClass.feedbackPin); // get
position of servo at end of defined servoTime (=motorTime) from pot

 aPos3R1[2] = motorClass.eTicks * -1 * 25.4 / (400 * 24); // converts
encoder ticks to Z of bar centroid relative to eye midpoint in mm
 //alphaServo = 187 - finalPosition;
 //rXY = sqrt(sq(servoClass.radius)-sq(aPos3R1[2]));
 //aPos3R1[1] = rXY * cos(alphaServo);
 aPos3R1[1] = sqrt(sq(servoClass.radius)-sq(aPos3R1[2])) * cos((187 -
finalPosition)* M_PI / 180); // Y of bar centroid relative to eye
midpoint in mm
 aPos3R1[0] = -sqrt(sq(servoClass.radius)-sq(aPos3R1[1])-
sq(aPos3R1[2])); // X of bar centroid relative to eye midpoint in mm


 // (B) FIND ACTUAL POINT LOOKING AT RELATIVE TO BASE FRAME

 aPos2R0[0]=parameterClass.pos2R0[0]; //input X same as actual X of
gripper relative to base
 aPos2R1[0]=parameterClass.pos2R1[0]; //input X same as actual X of
gripper relative to eye midpoint

 aPos3R0[0]= aPos3R1[0]+ parameterClass.pos1R0[0]; //get X,Y,Z of bar
centroid relative to base for calculations
 aPos3R0[1]= aPos3R1[1]+ parameterClass.pos1R0[1];
 aPos3R0[2]= aPos3R1[2]+ parameterClass.pos1R0[2];

 aMagA = aPos2R1[0] / ((parameterClass.pos1R0[0]-
aPos3R0[0])/servoClass.radius);

 aPos2R1[1] = aMagA * ((parameterClass.pos1R0[1]-
aPos3R0[1])/servoClass.radius);
 aPos2R1[2] = aMagA * ((parameterClass.pos1R0[2]-
aPos3R0[2])/servoClass.radius);
```

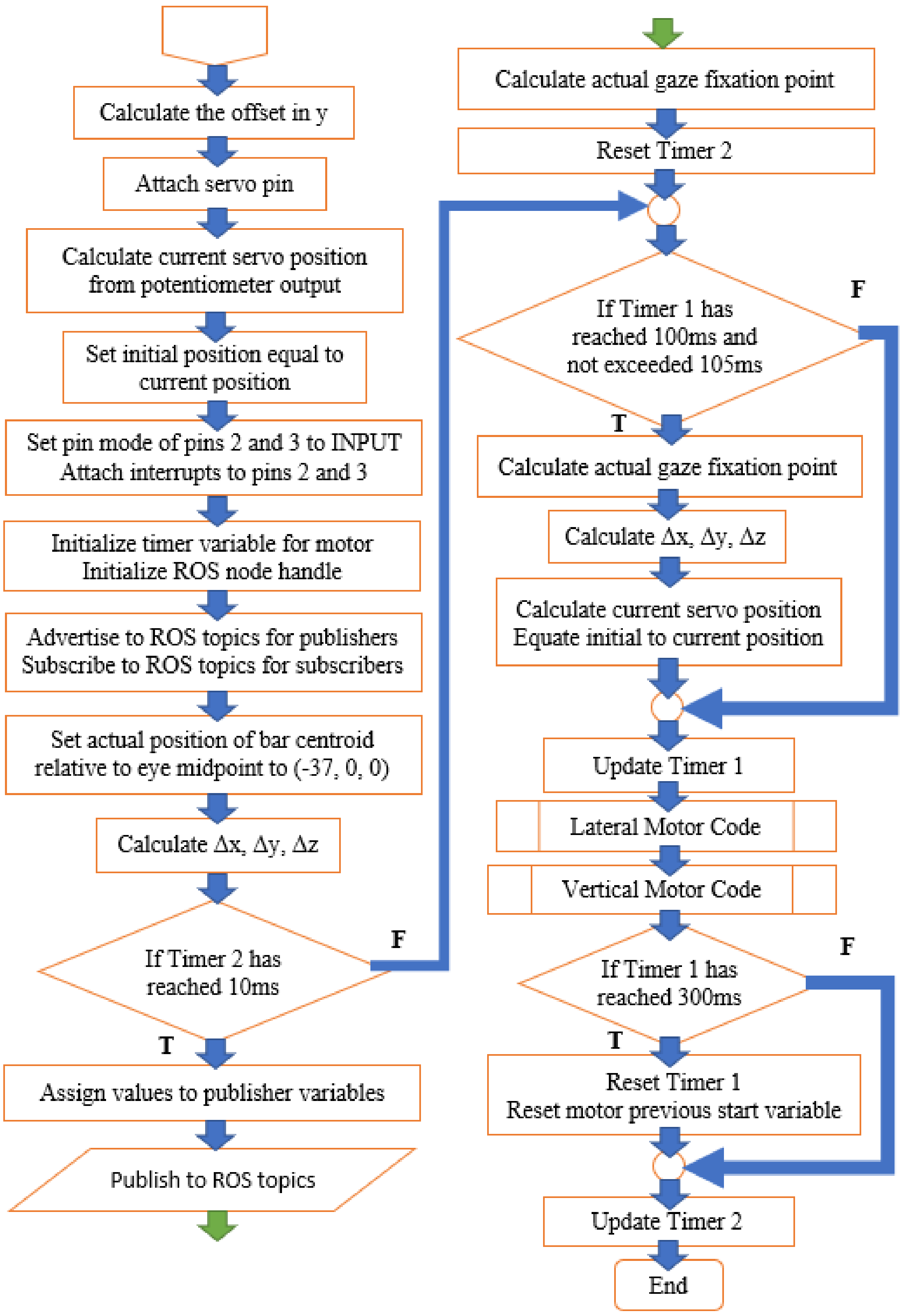
```
 aPos2R0[1]= aPos2R1[1]+ parameterClass.pos1R0[1]; //actual gripper Y
relative to base
 aPos2R0[2]= aPos2R1[2]+ parameterClass.pos1R0[2]; //actual gripper Z
relative to base

}

vClass verificationClass = vClass();
```

verification.h

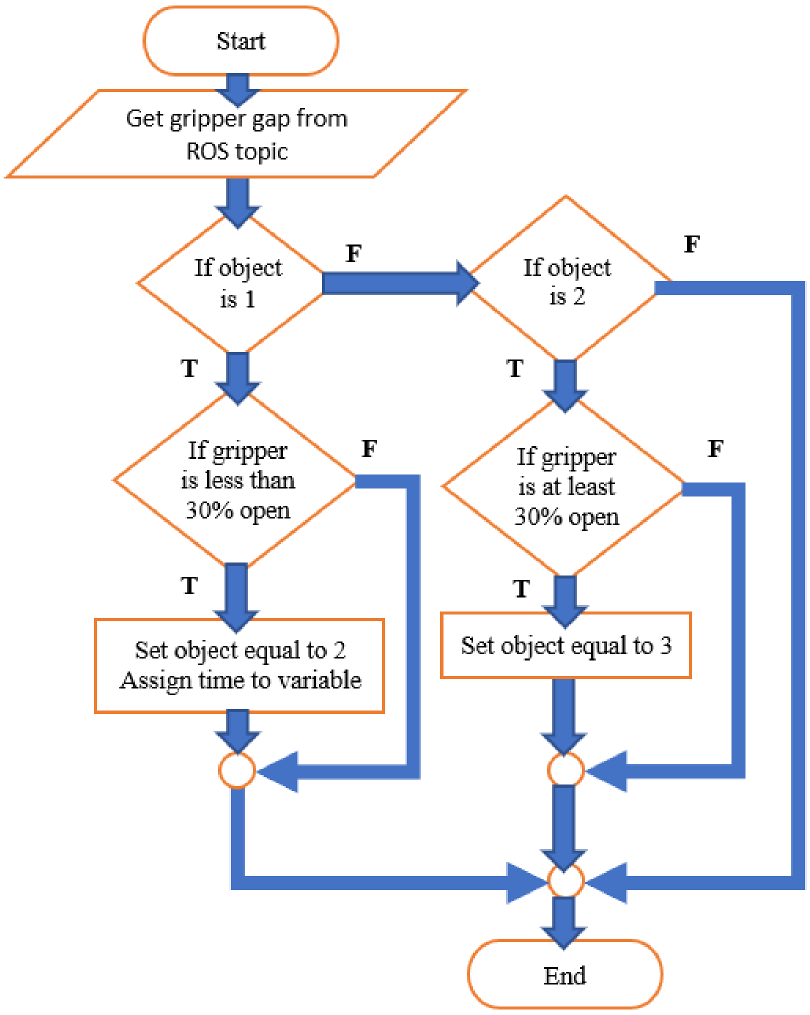
```
#ifndef verification_h
#define verification_h

#include <math.h>
class vClass
{
  public:
    vClass();
    void SETUP();
    void homeSetup();
    float finalPosition; //final servo position after servoTime
    float aPos3R1[3]; //actual bar centroid relative to eye midpoint
    float aPos2R1[3]; //actual gripper position relative to eye midpoint
    float aPos2R0[3]; //actual gripper position relative to base frame
    float aMagA; //actual magnitude of vector, A, from {1} to {2}
    float aPos3R0[3];
};

extern vClass verificationClass;

#endif
```

Arduino IDE Pseudocode

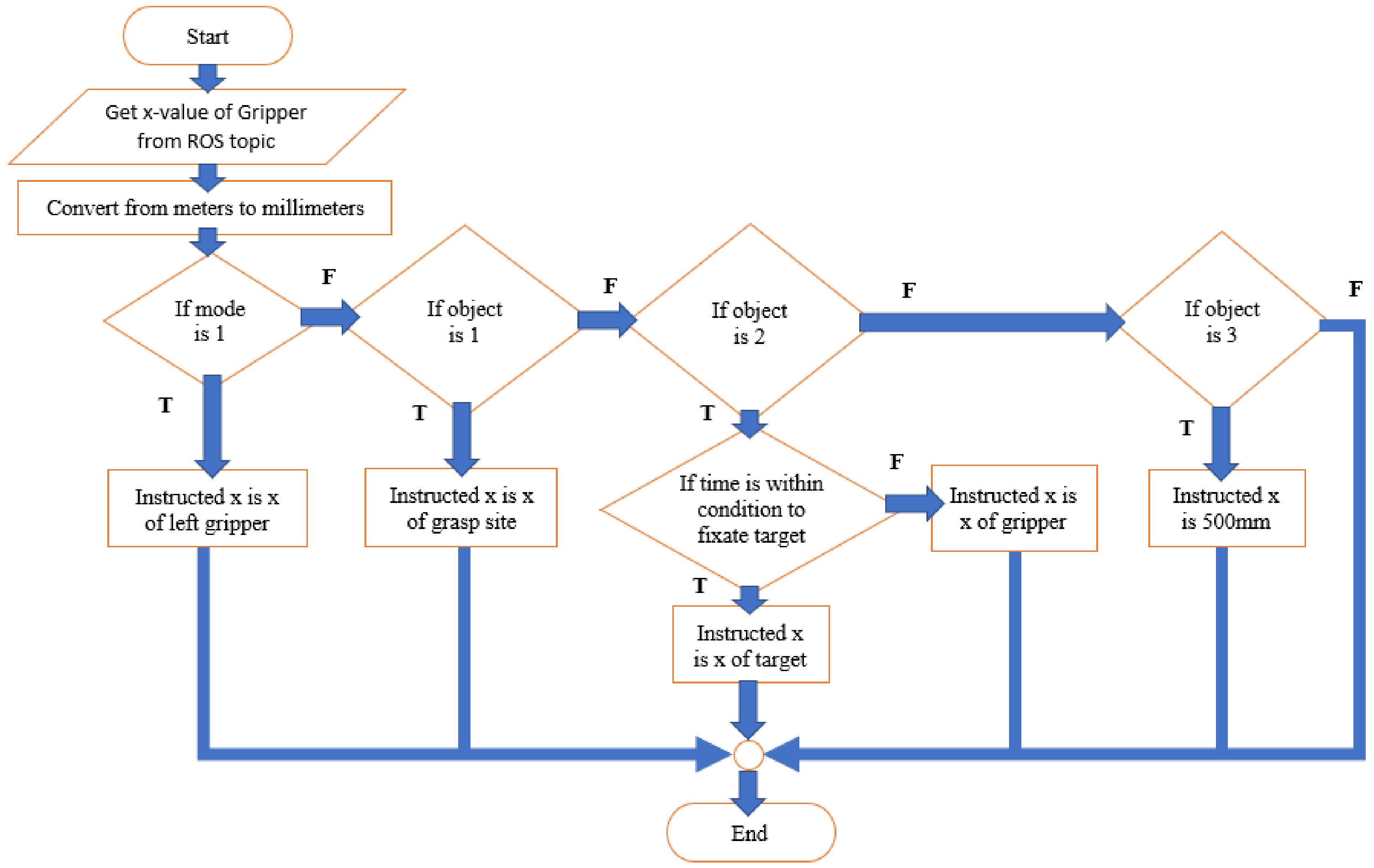

Start
Include 4 .h files for DH parameters, servo & pot, motor & encoder, verification
Include ros.h
Include Float32.h
Initialize 4 time keeping variables:
Start times for Timer 1 & Timer 2, current time for Timer 2, and variable for stage timer
Initialize stage variable "object" as 1
Assign 1, 2, or 3 to mode
Assign initial gaze fixation point, grasp site, target
Initialize variable for gripper gap
Initialize mode-dependent variables
Assign value to average velocity in y
Create shortcut for ROS node
Calculating gaze fixation point from gripper position
If mode is 2 (block)
F
If mode is 3 (full glass)
F
T
Set interval to look at target to 5380ms
Set temporal offset to 1500ms
Set condition to look away from target to 1300ms
T
Set interval to look at target to 3510ms
Set temporal offset to 1000ms
Set condition to look away from target to 1755ms


Calculate the offset in y
Attach servo pin
Calculate current servo position from potentiometer output
Set initial position equal to current position
Set pin mode of pins 2 and 3 to INPUT
Attach interrupts to pins 2 and 3
Initialize timer variable for motor
Initialize ROS node handle
Advertise to ROS topics for publishers
Subscribe to ROS topics for subscribers
Set actual position of bar centroid relative to eye midpoint to (-37, 0, 0)
Calculate Δx, Δy, Δz
If Timer 2 has reached 10ms
F
T
Assign values to publisher variables
Publish to ROS topics
Calculate actual gaze fixation point
Reset Timer 2
If Timer 1 has reached 100ms and not exceeded 105ms
F
T
Calculate actual gaze fixation point
Calculate Δx, Δy, Δz
Calculate current servo position
Equate initial to current position
Update Timer 1
Lateral Motor Code
Vertical Motor Code
If Timer 1 has reached 300ms
F
T
Reset Timer 1
Reset motor previous start variable
Update Timer 2
End

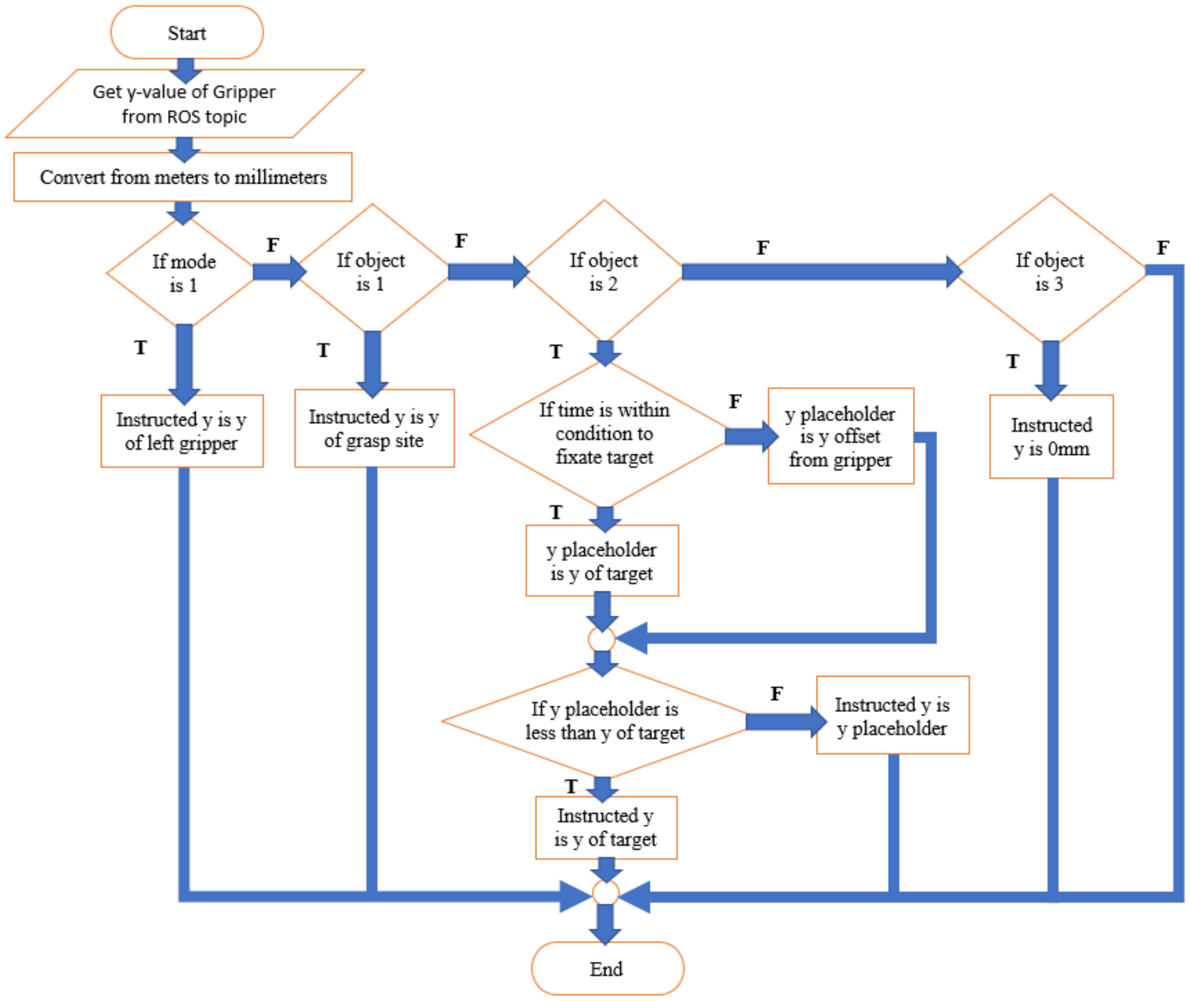
Arduino message function for gap between Baxter's fingertips


Start
Get gripper gap from ROS topic
If object is 1
F
If object is 2
F
T
T
If gripper is less than 30% open
F
If gripper is at least 30% open
F
T
T
Set object equal to 2 Assign time to variable
Set object equal to 3
End

Arduino message function for x of Baxter gripper

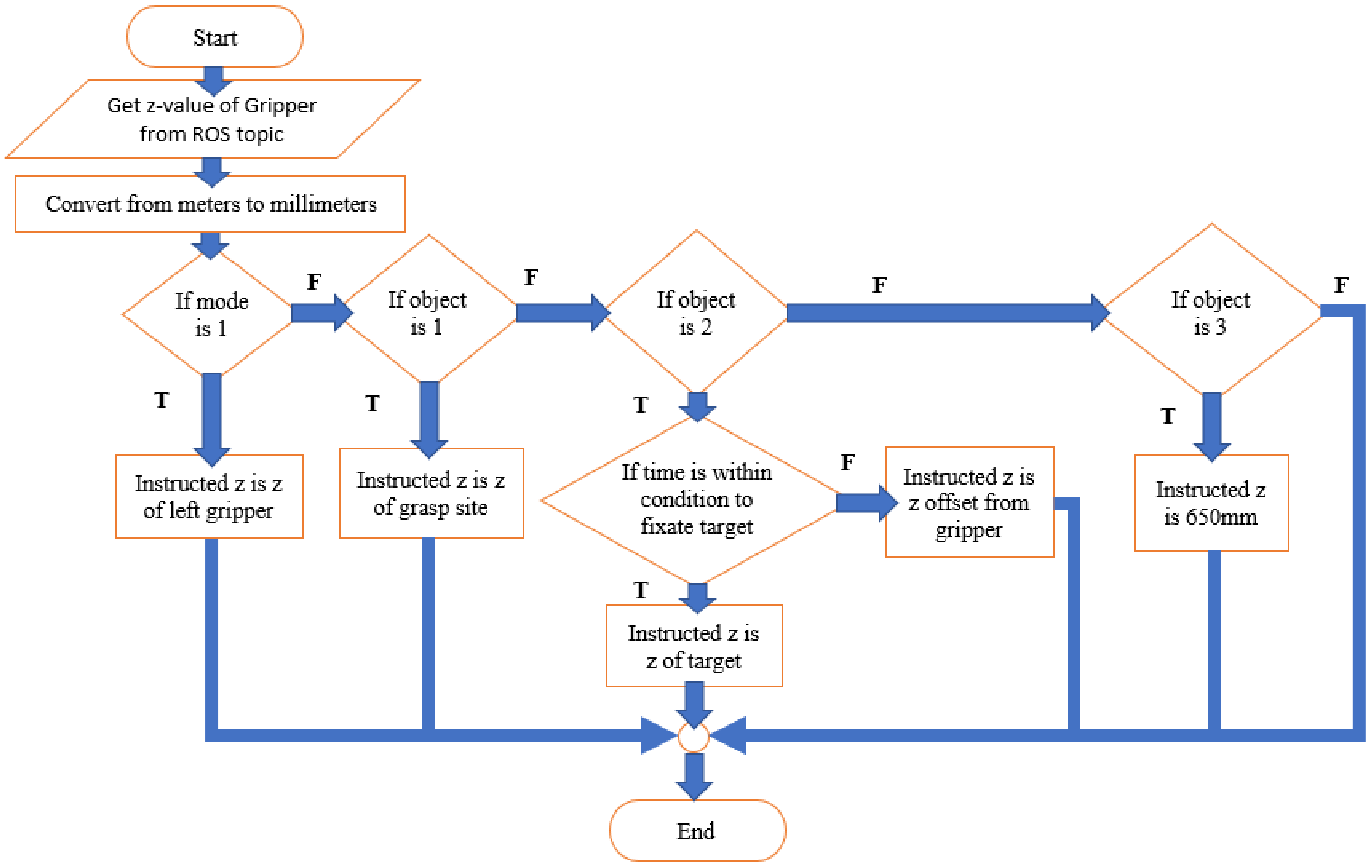

Start
Get x-value of Gripper from ROS topic
Convert from meters to millimeters
If mode is 1
F
T
Instructed x is x of left gripper
If object is 1
F
T
Instructed x is x of grasp site
If object is 2
F
T
If time is within condition to fixate target
F
T
Instructed x is x of gripper
Instructed x is x of target
If object is 3
F
T
Instructed x is 500mm
End

Arduino message function for y of Baxter gripper

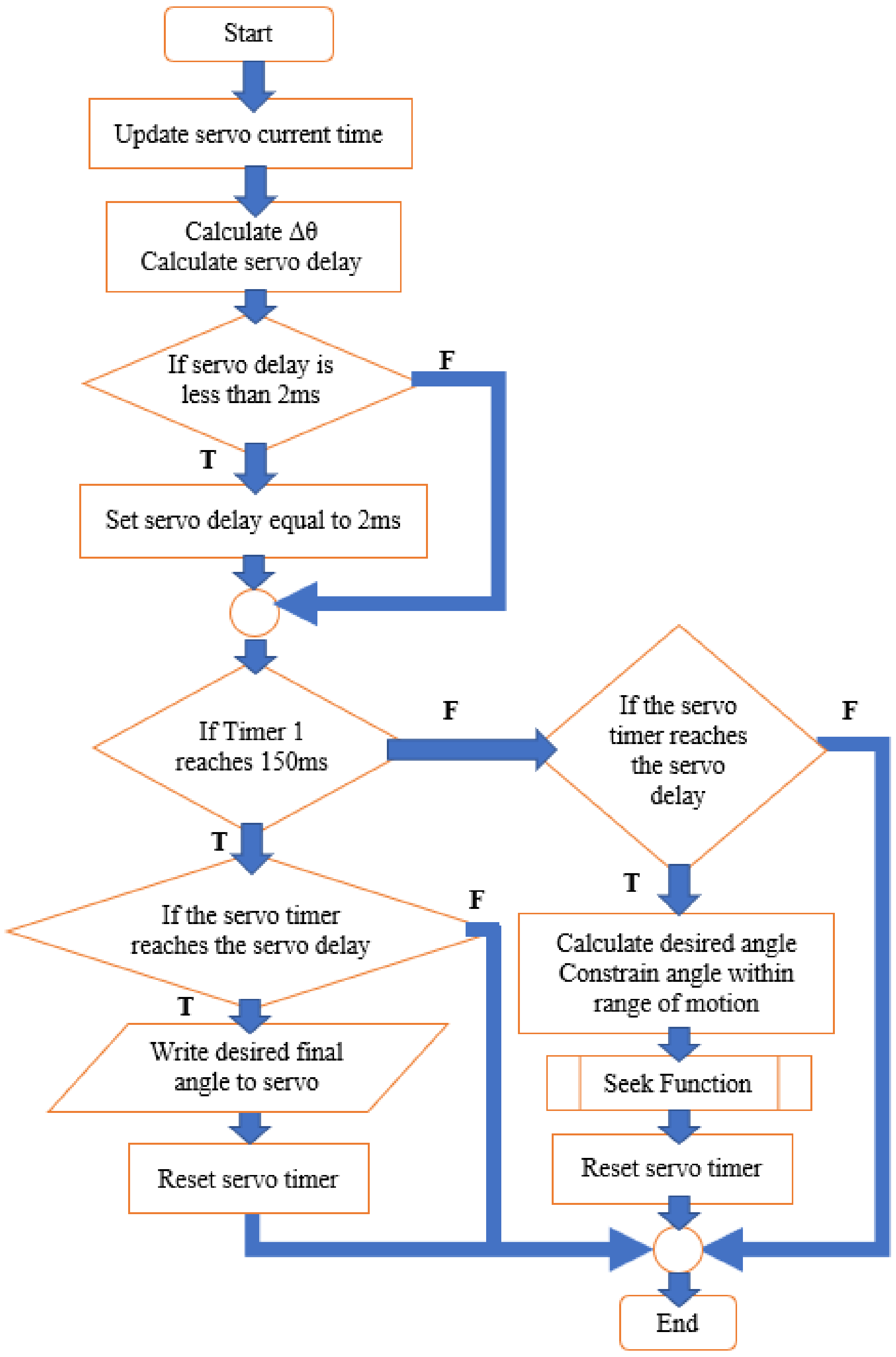


Start
Get y-value of Gripper from ROS topic
Convert from meters to millimeters
If mode is 1
F
T
Instructed y is y of left gripper
If object is 1
F
T
Instructed y is y of grasp site
If object is 2
F
T
If time is within condition to fixate target
F
y placeholder is y offset from gripper
T
y placeholder is y of target
If y placeholder is less than y of target
F
Instructed y is y placeholder
T
Instructed y is y of target
If object is 3
F
T
Instructed y is 0mm
End

Arduino message function for z of Baxter gripper

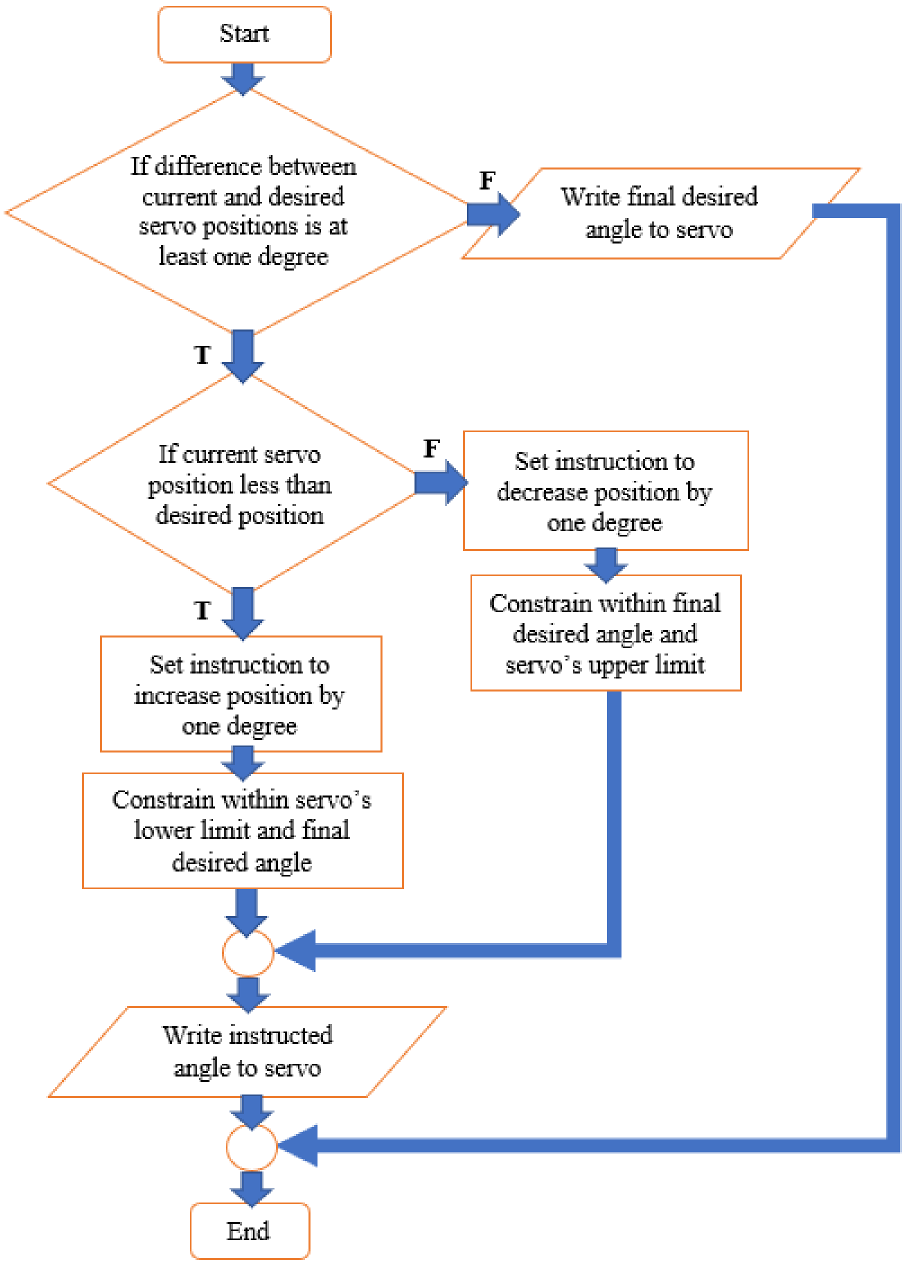

Start
Get z-value of Gripper from ROS topic
Convert from meters to millimeters
If mode is 1
F
T
Instructed z is z of left gripper
If object is 1
F
T
Instructed z is z of grasp site
If object is 2
F
T
If time is within condition to fixate target
F
Instructed z is z offset from gripper
T
Instructed z is z of target
If object is 3
F
T
Instructed z is 650mm
End

Lateral Motor Code (referenced in Arduino IDE Pseudocode)

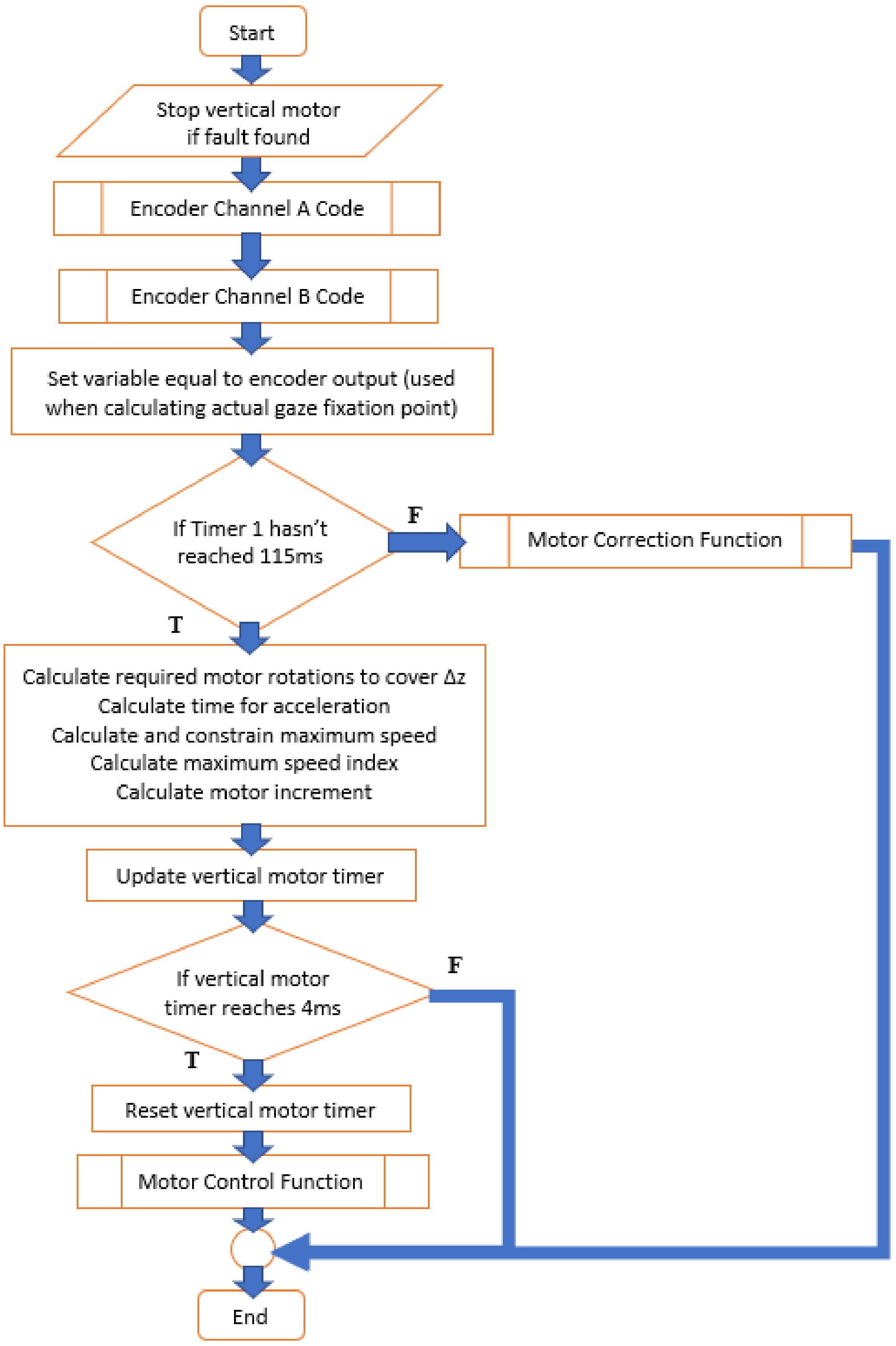


Start
Update servo current time
Calculate Δθ
Calculate servo delay
If servo delay is less than 2ms
F
T
Set servo delay equal to 2ms
If Timer 1 reaches 150ms
F
T
If the servo timer reaches the servo delay
F
T
Write desired final angle to servo
Reset servo timer
If the servo timer reaches the servo delay
F
T
Calculate desired angle
Constrain angle within range of motion
Seek Function
Reset servo timer
End

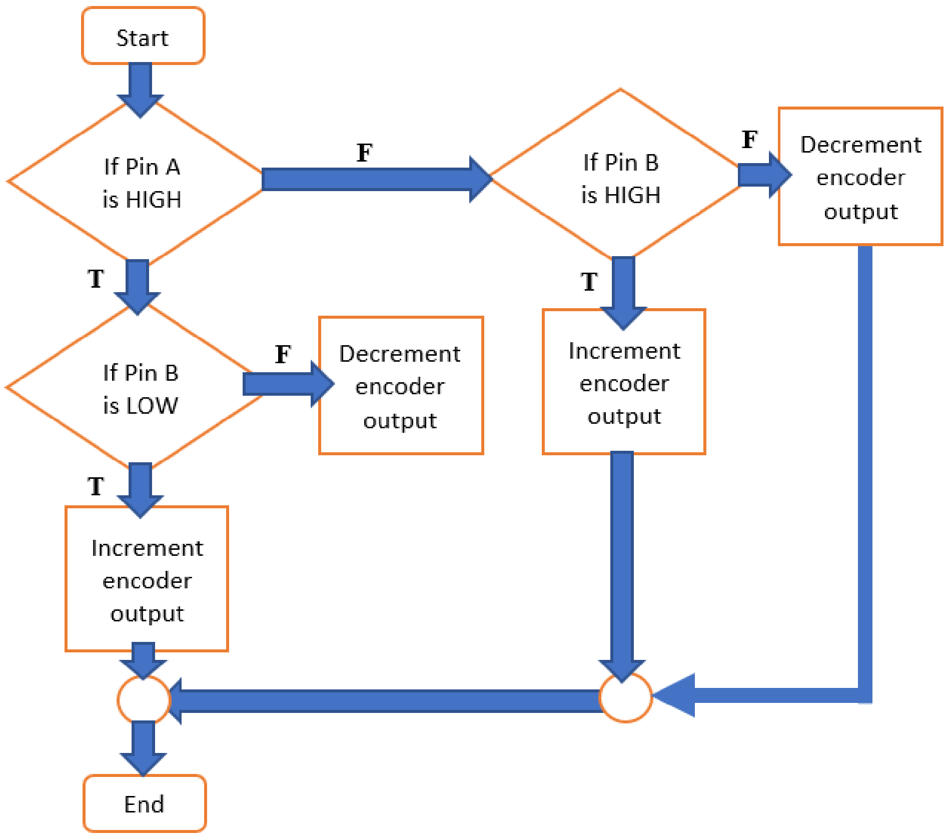
Seek Function (referenced in Lateral Motor Code)


Start
If difference between current and desired servo positions is at least one degree
F
Write final desired angle to servo
T
If current servo position less than desired position
F
Set instruction to decrease position by one degree
Constrain within final desired angle and servo's upper limit
T
Set instruction to increase position by one degree
Constrain within servo's lower limit and final desired angle
Write instructed angle to servo
End

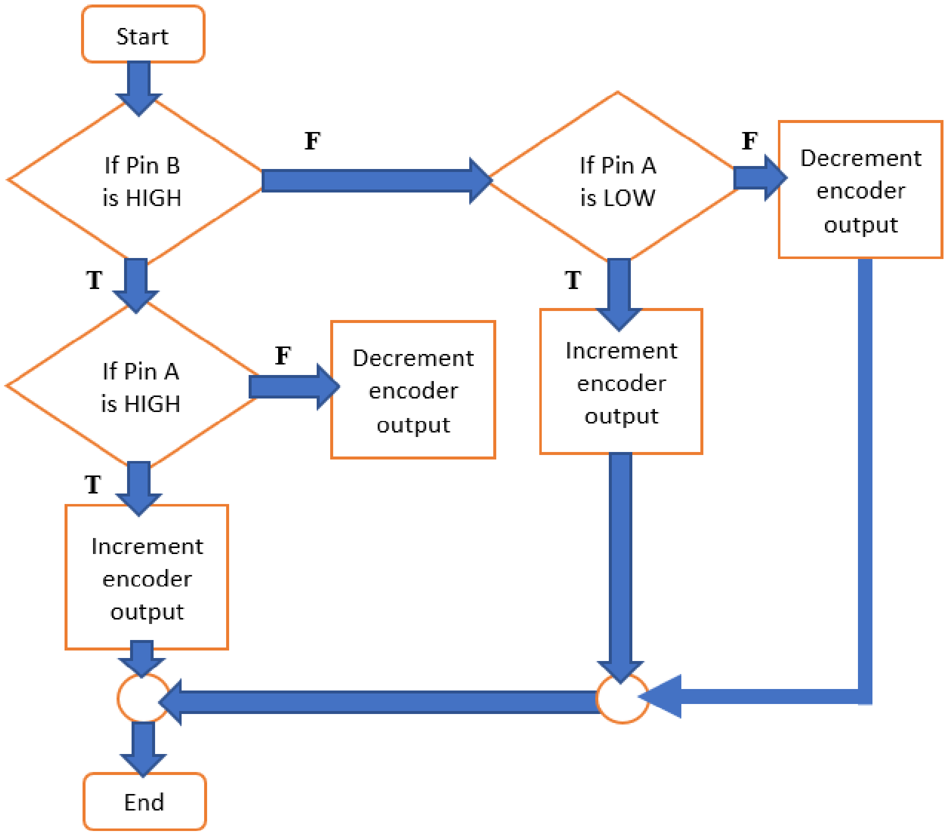
Vertical Motor Code (referenced in Arduino IDE Pseudocode)


Start
Stop vertical motor if fault found
Encoder Channel A Code
Encoder Channel B Code
Set variable equal to encoder output (used when calculating actual gaze fixation point)
If Timer 1 hasn't reached 115ms
F
Motor Correction Function
T
Calculate required motor rotations to cover Δz
Calculate time for acceleration
Calculate and constrain maximum speed
Calculate maximum speed index
Calculate motor increment
Update vertical motor timer
If vertical motor timer reaches 4ms
F
T
Reset vertical motor timer
Motor Control Function
End

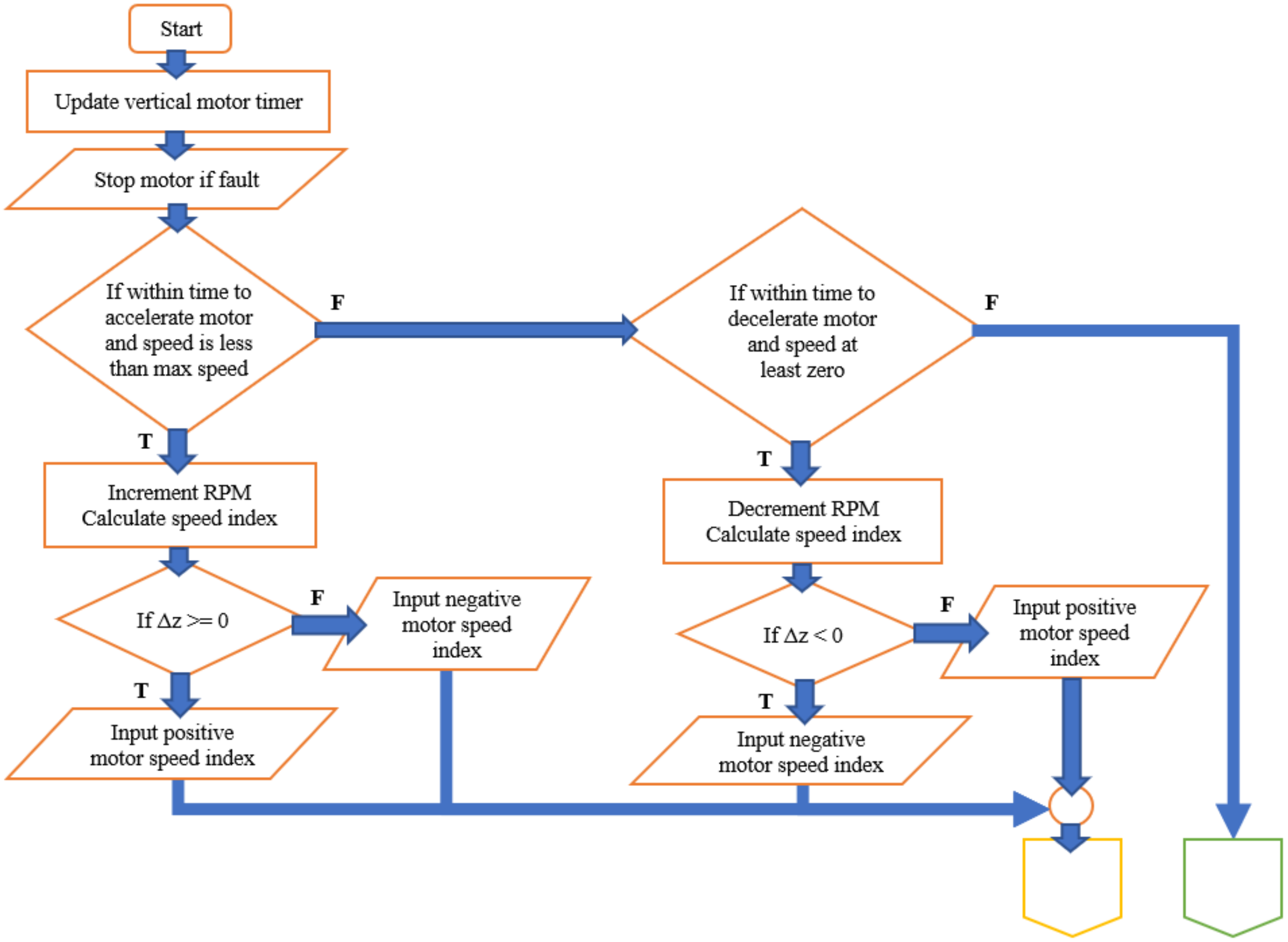
Encoder Channel A Code (referenced in Vertical Motor Code)

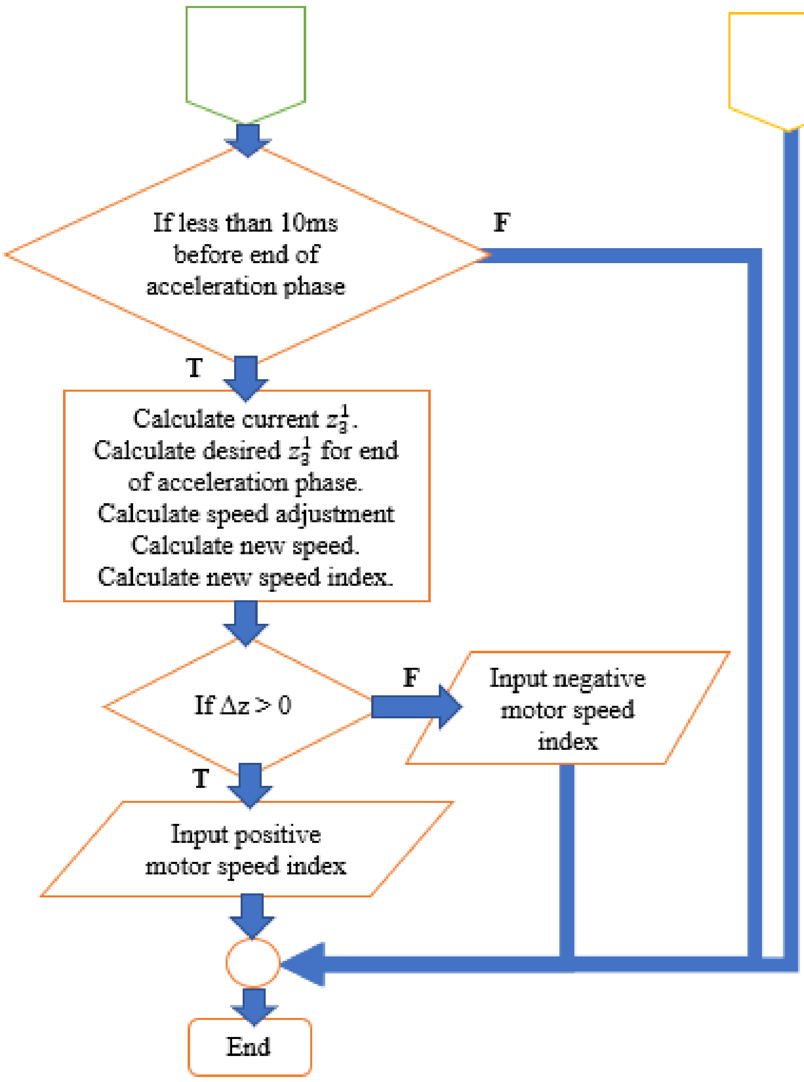


Encoder Channel B Code (referenced in Vertical Motor Code)

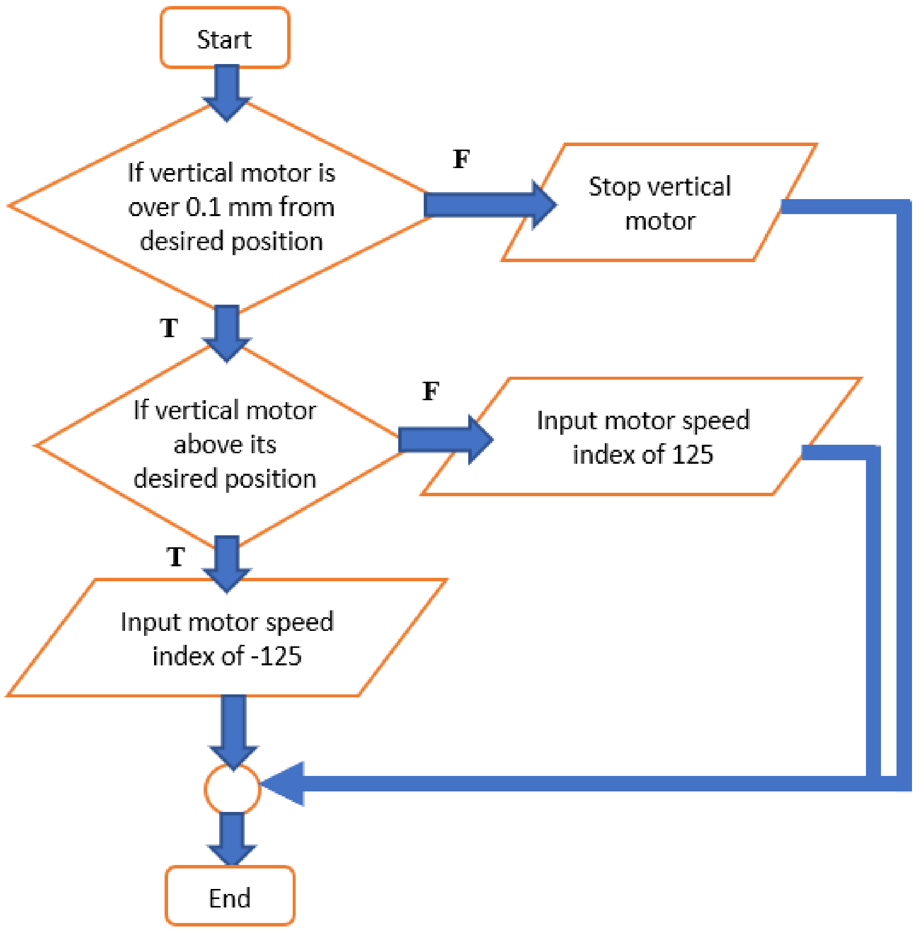


Motor Control Function (referenced in Vertical Motor Code)


Start
Update vertical motor timer
Stop motor if fault
If within time to accelerate motor and speed is less than max speed
F
T
Increment RPM Calculate speed index
If Δz >= 0
F
Input negative motor speed index
T
Input positive motor speed index
If within time to decelerate motor and speed at least zero
F
T
Decrement RPM Calculate speed index
If Δz < 0
F
Input positive motor speed index
T
Input negative motor speed index


If less than 10ms before end of acceleration phase
F
T
Calculate current z_3^1.
Calculate desired z_3^1 for end of acceleration phase.
Calculate speed adjustment
Calculate new speed.
Calculate new speed index.
If $\Delta z > 0$
F
Input negative motor speed index
T
Input positive motor speed index
End

Motor Correction Function (referenced in Vertical Motor Code)


Start
If vertical motor is over 0.1 mm from desired position
F
Stop vertical motor
T
If vertical motor above its desired position
F
Input motor speed index of 125
T
Input motor speed index of -125
End

Appendix E: Analog Read Code

```
int minDegrees=62;
int maxDegrees=132;
int minFeedback=725;
int maxFeedback=962;
int feedbackPin = A5;

int angle;

int getPos(int analogPin)
{
  return map(analogRead(analogPin), minFeedback, maxFeedback, minDegrees,
maxDegrees);
}

void setup()
{
  angle = analogRead(A5);
  Serial.begin(57600);
}

void loop() {
  // put your main code here, to run repeatedly:
//  Serial.println(angle); // prints analog reading on one line
  if (millis() % 10 < 5)
  {
    Serial.println(getPos(feedbackPin)); // prints calculated angle on next
line
  }

}
```

12 REFERENCES

[1] D. A. Atchison, “Optics of the Human Eye,” *Ref. Modul. Mater. Sci. Mater. Eng.*, pp. 1–19, 2017.

[2] N. Pateromichelakis *et al.*, “Head-eyes system and gaze analysis of the humanoid robot Romeo,” *IEEE Int. Conf. Intell. Robot. Syst.*, no. April 2015, pp. 1374–1379, 2014.

[3] D. Guitton and M. Volle, “Gaze control in humans: eye-head coordination during orienting movements to targets within and beyond the oculomotor range.,” *J. Neurophysiol.*, vol. 58, no. 3, pp. 427–459, 1987.

[4] H. Misslisch, D. Tweed, and T. Vilis, “Neural constraints on eye motion in human eye-head saccades.,” *J. Neurophysiol.*, vol. 79, no. 2, pp. 859–869, 1998.

[5] Y. B. Bang, J. K. Paik, B. H. Shin, and C. Lee, “A three-degree-of-freedom anthropomorphic oculomotor simulator,” *Int. J. Control Autom. Syst.*, vol. 4, no. 2, pp. 227–235, 2006.

[6] A. T. Bahill, M. R. Clark, and L. Stark, “The main sequence, a tool for studying human eye movements,” *Math. Biosci.*, vol. 24, no. 3–4, pp. 191–204, 1975.

[7] R. S. Johansson, G. Westling, A. Bäckström, and R. Flanagan, “Eye – Hand Coordination in Object Manipulation,” *J. Neurosci.*, vol. 21, no. 17, pp. 6917–6932, 2001.

[8] A. M. Bronstein and C. Kennard, “Predictive eye saccades are different from visually triggered saccades,” *Vision Res.*, vol. 27, no. 4, pp. 517–520, 1987.

[9] D. A. Robinson, “The Mechanics of Human Saccadic Eye Movement,” *J. Physiol.*, pp. 245–264, 1964.

[10] H. Liu, J. Luo, P. Wu, S. Xie, and H. Li, “Symmetric kullback-leibler metric based tracking behaviors for bioinspired robotic eyes,” *Appl. Bionics Biomech.*, vol. 2015, 2015.

[11] C. M. Gosselin and J.-F. Hamel, “The agile eye: a high-performance three-degree-of-freedom camera-orienting device,” *Proc. 1994 IEEE Int. Conf. Robot. Autom.*, pp. 781–786, 1994.

[12] L. Ramos, S. Valencia, S. Verma, K. Zornoza, M. Morris, and S. Tosunoglu, “Robotic Face to Simulate Humans Undergoing Eye Surgery,” 2017.

[13] “Eye Mechanism | InMoov.” [Online]. Available: http://inmoov.fr/eye-mechanism/. [Accessed: 15-Apr-2018].

[14] “Eye modification for InMoov by bhouston - Thingiverse,” 2015. [Online]. Available: https://www.thingiverse.com/thing:1047175. [Accessed: 15-Apr-2018].

[15] K. Bassett, M. Hammond, and L. Smoot, “A fluid-suspension, electromagnetically driven eye with video capability for animatronic applications,” *9th IEEE-RAS Int. Conf. Humanoid Robot. HUMANOIDS09*, pp. 40–46, 2009.

[16] H. Irmler *et al.*, “United States Patent: System and method for generating realistic eyes,” US 8,651,916 B2, 2014.

[17] E. Brockmeyer, I. Poupyrev, and S. Hudson, “Papillon: Designing Curved Display Surfaces With Printed Optics,” *Proc. 26th Annu. ACM Symp. User interface Softw. Technol. - UIST ’13*, pp. 457–462, 2013.

[18] C. D. Kidd and C. Breazeal, “Effect of a robot on user perceptions,” *2004 IEEE/RSJ Int. Conf. Intell. Robot. Syst. (IEEE Cat. No.04CH37566)*, vol. 4, no. January 2004, pp. 3559–3564, 2004.

[19] T. Landgraf, D. Bierbach, H. Nguyen, N. Muggelberg, P. Romanczuk, and J. Krause, “RoboFish: Increased acceptance of interactive robotic fish with realistic eyes and natural motion patterns by live Trinidadian guppies,” *Bioinspiration and Biomimetics,* vol. 11, no. 1, p. 15001, 2016.

[20] J. Ham, R. H. Cuijpers, and J. J. Cabibihan, “Combining Robotic Persuasive Strategies: The Persuasive Power of a Storytelling Robot that Uses Gazing and Gestures,” *Int. J. Soc. Robot.*, vol. 7, no. 4, pp. 479–487, 2015.

[21] Aj. Moon *et al.*, “Meet me where i’m gazing: How shared attention gaze affects human-robot handover timing,” *Proc. 2014 ACM/IEEE Int. Conf. Human-robot Interact. - HRI ’14*, no. March, pp. 334–341, 2014.

[22] C. J. Stanton and C. J. Stevens, “Don’t Stare at Me: The Impact of a Humanoid Robot’s Gaze upon Trust During a Cooperative Human–Robot Visual Task,” *Int. J. Soc. Robot.*, vol. 9, no. 5, pp. 745–753, 2017.

[23] M. Land, N. Mennie, and J. Rusted, “The roles of vision and eye movements in the control of activities of daily living,” *Perception*, vol. 28, no. 11, pp. 1311–1328, 1999.

[24] M. Hayhoe, “Vision Using Routines: A Functional Account of Vision,” *Vis. cogn.*, vol. 7, no. 1–3, pp. 43–64, 2000.

[25] M. F. Land and M. M. Hayhoe, “In what ways do eyemovements contribute to everyday activities?,” *Vision Res.*, vol. 41, no. 25–26, pp. 3559–3565, 2001.

[26] S. T. Iqbal and B. P. Bailey, “Using Eye Gaze Patterns to Identify User Tasks,” *Grace Hopper Celebr. Women Comput.*, p. 6, 2004.

[27] T. W. Victor, J. L. Harbluk, and J. A. Engström, “Sensitivity of eye-movement measures to in-vehicle task difficulty,” *Transp. Res. Part F Traffic Psychol. Behav.*, vol. 8, no. 2 SPEC. ISS., pp. 167–190, 2005.

[28] S. Fitzgerald and M. Shiloh, *The Arduino Projects Book*, 3rd ed. Torino, 2015.

[29] “Pololu Dual MC33926 Motor Driver Shield for Arduino.” [Online]. Available: https://www.pololu.com/product/2503. [Accessed: 15-Apr-2018].

[30] “Workstation Setup - sdk-wiki.” [Online]. Available: http://sdk.rethinkrobotics.com/wiki/Workstation_Setup. [Accessed: 15-Apr-2018].

[31] “Baxter PyKDL - sdk-wiki.” [Online]. Available: http://sdk.rethinkrobotics.com/wiki/Baxter_PyKDL. [Accessed: 15-Apr-2018].

[32] “Joint Trajectory Playback Example - sdk-wiki.” [Online]. Available: http://sdk.rethinkrobotics.com/wiki/Joint_Trajectory_Playback_Example. [Accessed: 22-Jun-2018].

[33] “Baxer User Guide for Intera 3.0 Software,” 2014. [Online]. Available: http://mfg.rethinkrobotics.com/mfg-mediawiki-1.22.2/images/1/12/Baxter_User_Guide_for_Intera_3.0.0.pdf.

[34] “Arduino Reference map().” [Online]. Available: https://www.arduino.cc/reference/en/language/functions/math/map/. [Accessed: 22-Jun-2018].

[35] “Servo library.” [Online]. Available: https://www.arduino.cc/en/Reference/Servo. [Accessed: 15-Apr-2018].

[36] D. J. Gonzalez, “YAMEB: Quadrature Encoders in Arduino, done right. Done right.,” 2012. [Online]. Available: http://yameb.blogspot.com/2012/11/quadrature-encoders-in-arduino-done.html. [Accessed: 15-Apr-2018].

[37] “Arduino library for the Pololu Dual MC33926 Motor Driver Shield.” [Online]. Available: https://github.com/pololu/dual-mc33926-motor-shield. [Accessed: 15-Apr-2018].

[38] “API Reference - sdk-wiki.” [Online]. Available: http://sdk.rethinkrobotics.com/wiki/API_Reference. [Accessed: 15-Apr-2018].

[39] “Gripper Customization - sdk-wiki.” [Online]. Available: http://sdk.rethinkrobotics.com/wiki/Gripper_Customization. [Accessed: 15-Apr-2018].

[40] “Arduino Playground - RotaryEncoders.” [Online]. Available: https://playground.arduino.cc/Main/RotaryEncoders. [Accessed: 15-Apr-2018].

[41] “ROS + Arduino = Robot.” [Online]. Available: https://github.com/hbrobotics/ros_arduino_bridge.