Lab 5: Final Project

Cory Brolliar, Clayton Dembski, Zoraver Kang

I. INTRODUCTION

The task of this final project was to use our 3-DOF arm along with our camera to identify the objects by color, acquire them and weigh them, then sort them by both color and weight. To determine the weight of each object, we used the load cells that were provided on the arm. This final project ties all of the components from all of the individual labs together into an automated robotic sorting system.

II. METHODOLOGY

A. Communication Protocol

To enable communication between our arm and a lab computer running MATLAB, we built on the framework provided to us. On the MATLAB side, we utilize the provided `PacketProcessor` class with wrapper functions to make our code more concise. On the arm side, we utilized several classes which extend `PacketEventAbstract` in a way similar to the approach taken in `PidServer`. These classes are `StatusServer`, `GainServer`, and `ClawServer`.

`PidServer` handles the `SET_PID` command (ID = 37). This command sets the setpoints of the PID controllers of the arm based off of the given arguments. Its arguments are:

1) Position setpoint for axis 0 (base axis) in raw encoder ticks
2) Velocity target for axis 0 (base axis) in encoder ticks per second
3) Force target for axis 0
4) Position setpoint for axis 1 (shoulder) in raw encoder ticks
5) Velocity target for axis 1 (shoulder) in encoder ticks per second
6) Force target for axis 1
7) Position setpoint for axis 2 (elbow) in raw encoder ticks
8) Velocity target for axis 2 (elbow) in encoder ticks per second
9) Force target for axis 2

The velocity and force target systems are not currently implemented. The `SET_PID` command returns the the following values: (position of encoder on axis 0, 0 (reserved), 0 (reserved), position of encoder on axis 1, 0 (reserved), 0 (reserved), position of encoder on axis 2, 0 (reserved), 0 (reserved)).

`StatusServer` handles the `STATUS` command (ID = 38). This command takes no arguments and returns the current values of the internal sensors of the arm is the following order:

1) Position of encoder 0 (base axis) in raw encoder ticks
2) Position of encoder 1 (shoulder) in raw encoder ticks
3) Position of encoder 2 (elbow) in raw encoder ticks
4) Velocity measured by encoder 0 in encoder ticks per second
5) Velocity measured by encoder 1 in encoder ticks per second
6) Velocity measured by encoder 2 in encoder ticks per second
7) Base axis torque sensor ADC reading, averaged over 5 samples
8) Shoulder torque sensor ADC reading, averaged over 5 samples
9) Elbow torque sensor ADC reading, averaged over 5 samples

`GainServer` handles the `SET_GAIN` command (ID = 39). This command allows for the PID coefficients of any of the PID controllers on the device to have their gains set via MATLAB. The arguments of this command are (zero-indexed axis number corresponding to the PID controller you want to configure, new k_p, new k_i, new k_d). Before changing the gains of the specified PID controller, the `GainServer` disables that specific controller. Once the gains are updated, it is re-enabled. The `SET_GAIN` command does not provide any return values.

`ClawServer` handles the `SET_CLAW` command (ID = 40). This command allows for the claw at the tip of the arm to be controlled via MATLAB. It takes a single argument: a float between 0.0 and 1.0 which corresponds to the position of the servo within its

overall range of motion. The `SET_CLAW` command does not provide any return values.

B. Inverse Position Kinematics

The inverse position kinematics of arm were determined through the use of geometry and trigonometry. By projecting the tip of the arm onto the xy plane, we easily found $q_0 = atan(p_y/p_x)$. In Matlab, we use the `atan2()` function to avoid the edge cases of the `atan()` function, so the above expression for q_0 becomes `q0 = atan2(x, y)`.

To determine the values of q_1 and q_2, we examined the arm in the plane shared by all three links, as shown in figure 11. The value of H can be easily determined using the Pythagorean theorem.

$$H = \sqrt{p_x^2 + p_y^2 + (p_z - L_1)^2}$$

Now that the value of H is known, Θ can be determined using the law of cosines.

$$H^2 = L_2^2 + L_3^2 - 2L_2L_3cos(\Theta)$$

$$2L_2L_3cos(\Theta) = L_2^2 + L_3^2 - H^2$$

$$cos(\Theta) = \frac{L_2^2 + L_3^2 - H^2}{2L_2L_3}$$

$$\Theta = acos\left(\frac{L_2^2 + L_3^2 - H^2}{2L_2L_3}\right)$$

Using the value of Θ, q_2 can be determined.

$$q_2 = \pi - \frac{\pi}{2} - \Theta = \frac{\pi}{2} - \Theta$$

The values of H and Θ can be used in conjunction with the law of sines to determine the value of Φ.

$$\frac{sin(\Phi)}{L_3} = \frac{sin(\Theta)}{H}$$

$$\Phi = asin\left(\frac{L_3}{H}sin(\Theta)\right)$$

The value of q_1 can be determined through the use of geometry and the value of Φ.

$$q_1 = \Phi + atan\left(\frac{p_z - L_1}{\sqrt{p_x^2 + p_y^2}}\right) = asin\left(\frac{L_3}{H}sin(\Theta)\right) + atan\left(\frac{p_z - L_1}{\sqrt{p_x^2 + p_y^2}}\right)$$

$$= asin\left(\frac{L_3}{\sqrt{(p_x^2 + p_y^2 + (p_z - L_1)^2)}}sin\left(acos\left(\frac{L_2^2 + L_3^2 - (p_x^2 + p_y^2 + (p_z - L_1)^2)}{2L_2L_3}\right)\right)\right)$$

$$+atan\left(\frac{p_z - L_1}{\sqrt{p_x^2 + p_y^2}}\right)$$

In summary, the inverse position kinematics of our 3-DOF arm are:

$$q_0 = atan\left(\frac{p_y}{p_x}\right)$$

$$q_1 = asin\left(\frac{L_3}{\sqrt{(p_x^2 + p_y^2 + (p_z - L_1)^2)}}sin\left(acos\left(\frac{L_2^2 + L_3^2 - (p_x^2 + p_y^2 + (p_z - L_1)^2)}{2L_2L_3}\right)\right)\right)$$

$$+atan\left(\frac{p_z - L_1}{\sqrt{p_x^2 + p_y^2}}\right)$$

$$q_2 = \frac{\pi}{2} - acos\left(\frac{L_2^2 + L_3^2 - (p_x^2 + p_y^2 + (p_z - L_1)^2)}{2L_2L_3}\right)$$

C. Trajectory Generation Function

To implement trajectory generation, the function `TrajectoryGen.m` was created matching the method using position and velocity. The function accepted 2 times, 2 positions, 2 velocities and a time step and returned a time array, a position array, a velocity array and an acceleration array. The function was to match a cubic polynomial for position dynamics:

$$P_{dir} = a_0 + a_1 * t + a_2 * t^2 + a_3 * t^3$$

As this equation was mapped as a 1 dimensional vector with respect to time, three polynomials were needed, one for travel in the x direction, one for travel in the y direction, and one for travel in the z direction. For velocity dynamics, the derivative could be taken:

$$V_{dir} = a_1 + 2 * a_2 * t + 3 * a_3 * t^2$$

For the 3rd return data, acceleration, the derivative could be taken one more time:

$$V_{dir} = 2 * a_2 + 6 * a_3 * t$$

Given 4 points, 2 positions, initial and final, and 2 velocities, initial and final, and a time period for this to take place, the cubic could be solved through linear algebra:

$$\begin{bmatrix} 1 & t_i & t_i^2 & t_i^3 \\ 0 & 1 & 2*t_i & 3*t_i^2 \\ 1 & t & t_f^2 & t_f^3 \\ 0 & 1 & 2*t_f & 3*t_f^2 \end{bmatrix} * \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} p_i \\ v_i \\ p_f \\ v_f \end{bmatrix}$$

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 & t_i & t_i^2 & t_i^3 \\ 0 & 1 & 2*t_i & 3*t_i^2 \\ 1 & t & t_f^2 & t_f^3 \\ 0 & 1 & 2*t_f & 3*t_f^2 \end{bmatrix}^{-1} * \begin{bmatrix} p_i \\ v_i \\ p_f \\ v_f \end{bmatrix}$$

where time and p and v are the initial and final positions and velocities respectively.

Once each constant a_i was solved for each axis with respect to frame F_0, a series of points could be generated using the time step. An array of times, from the initial time to the final time was generated. Each time was plugged into the corresponding cubic, parabolic and linear equations to generate x y and z positions, velocities, and accelerations, all of which were then returned to the user.

In order to execute a trajectory, a set of task space position set points is generated using a trajectory generation function and a timer is set up to call a function which moves the arm to the given set point for the current timestep at a constant time interval. In our case, we used a time interval of 100 milliseconds. This function uses the `inversepk()` function to determine the joint angles for each setpoint and sets the position PID setpoint of the arm using the `setPosition()` function.

D. Joint Torque Sensor Calibration

In order to calibrate the joint torque sensors, we moved the arm into its fully vertical position (q_0 = 0, q_1 = $\pi/2$, q_2 = $-\pi/2$) with the motors unpowered and read each torque sensor five times. The zero offset for each joint torque sensor was determined by taking the average of the corresponding set of readings. The zero offsets are applied on the Nucleo side, in `StatusServer`. In addition, the `StatusServer` returns an average of the five most recent ADC readings for each joint torque sensor value.

We utilized the provided scale factors for the joint torque sensors. The given equation for this was $ADC = 178.5 * torque + 1918.4$. As this was a 12 bit ADC, to find the torque, we rearranged this equation to be $\frac{4096 - 1918.4}{178.5}$ or 12.199, our scale factor.

E. Displaying Applied Force Vectors on a Live Plot

After calibrating the joint torque sensors, the next task was to show, on the 3D plot, the force applied at any given time. First, we improved the 3D plot already displayed by the System, adding and labeling the frames at each joint. To show the frames for said joints, we added the function `mArrow3()` to plot a 3d arrow on our graph. To have them pointing in the direction of the x, y, and z axis of each frame, we were able to use the frame transformations multiplied by a constant of 5, to make them more visible. For each T_a^b where T is a 4 by 4 matrix, the first 3 columns represent the unit vector in the x, y, and z direction with respect to a. The frame at the end of link 1 was found using T_0^1, link 2 was T_0^2 and link 3 was T_0^3. The frame names were set to be a constant height above where the frame was set. Once we had this updated graph, the inverse Jacobian was used to identify the force and add said force to the tip of arm in the live plot. The Jacobian itself could be found in a function created for a previous lab: `inversevk()` and was given by figure 12. The torque at each joint of the arm is found as follows:

$$\begin{bmatrix}\tau_1\\ \tau_2\\ \tau_3\end{bmatrix} = \left[J(3x6)\right]^T * \begin{bmatrix}f_x\\ f_y\\ f_z\\ n_x\\ n_y\\ n_z\end{bmatrix}$$

When examining the arm when it is holding an object at its tip, the moments applied to the tip are minuscule and can be approximated to zero (the heavy object can rotate in the grip so that it is always exerting a force approximately straight down at any given instant). Because of this, the corresponding half of the angular Jacobian can be discarded, and the equation simplifies to:

$$\begin{bmatrix}\tau_1\\ \tau_2\\ \tau_3\end{bmatrix} = \left[J(3x3)\right]^T * \begin{bmatrix}f_x\\ f_y\\ f_z\end{bmatrix}$$

This transpose Jacobian is invertible, as the number of rows match the number of columns. Because of this, methods of linear algebra can be used to find the force vector.

$$\left[\left[J(3x6)\right]^T\right]^{-1} * \begin{bmatrix}\tau_1\\ \tau_2\\ \tau_3\end{bmatrix} = \begin{bmatrix}f_x\\ f_y\\ f_z\end{bmatrix}$$

Once we were able to generate the graph, we had to display the force vectors for 3 arbitrary conditions, seen by figures 1-8. The first graph in each set is the vector unloaded, while the second in each set is loaded with a net force of 5 newtons. From these graphs, it is apparent that the force is changing by a total of 5 newtons. It is also apparent that the plot can react to an applied force in any direction in 3 dimensional space, not just a force the arm is gripping (IE: in the negative Z direction with respect to frame F_0).

F. Live Object Measurement

To generate our live object measurement, we edited the `setpointsipk()` script a new created for a previous lab to create a new script, `setpointipk5()`. This script specifically took a series of set points and created a trajectory between each of them. In addition, using Matlab's timer library, the script created a series of set interrupts. Each time an interrupt was called, the current data for the arm, encoder positions, velocities and forces, were returned and appended to a csv file that could later be read from. The 10 set points were placed to be on either side of the robot, slowly increasing in the x direction with every cycle back and forth. The heavy weight was used to give the best force sensing results. Using this data, three graphs were generated: Position vs Time, Force vs Time, and Net Force vs Time. These graphs can be seen in figure 9.

G. Object Localization

In order to determine the positions of each object in the workspace, we developed a MATLAB function to detect the position from the viewpoint of the camera. This function `getAllTargetPosition()` takes two arguments: a camera object, and a number corresponding to a color (1 corresponds to the color of the blue object, 2 corresponds to the color of the green object, and 3 corresponds to the color of the yellow object). It returns the pixel coordinates of the centroid of the largest object of the specified color.

`getAllTargetPosition()` contains a vision pipeline as described below:

1) Acquire image from webcam
2) Crop the image to only show the workspace area
3) Convert the image from the RGB color space to the HSV color space
4) Threshold the image using the HSV description of the selected color
5) Fill holes in the resulting blobs
6) Extract the diameters and centroids of each blob

7) Select the blob with the largest diameter and return its centroid if the diameter is above a certain threshold

The thresholds for each color were determined using the MATLAB Color Thresholder app in the HSV color space. By carefully tuning these thresholds, we were able to eliminate the false positives for the yellow target caused by the wooden floor of the workspace that were initially present.

H. Automated Sorting System

In order to create the software which enabled the arm to automatically pick up and sort all of the objects in the workspace, we utilized our previous work on forward and inverse position kinematics, trajectory generation, force sensing, and vision processing. We delegated tasks between the lab computer running MATLAB and the Nucleo-controlled arm as follows. The MATLAB side handled the user interface, forward position kinematics, inverse position kinematics, trajectory generation, force propagation, vision processing, and the overall control logic. The Nucleo side handled the PID controllers and low-level interfaces to the encoders and joint torque sensors. Communication between the computer running MATLAB and the Nucleo was facilitated by the communication protocol described in section II-A.

A simplified version of the high-level control logic of the automated sorting system is as follows:

1) Open the claw
2) Move the arm in a smooth trajectory from its current position to a position where it does not block the camera
3) Using vision, scan until an object is found and store its color (continue) or the system times out (exit)
4) Move the arm using inverse position kinematics to a position 4" above the target object
5) Lower the claw onto the object using a smooth trajectory and close the claw
6) Raise the claw to a position 4" off the ground above its previous position using a smooth trajectory
7) Move the arm into its standard weight measurement position[1] using a smooth trajectory
8) Weigh the object and determine whether its base is light or heavy
9) Move the arm to a position determined based on the color and weight of the object using a smooth trajectory and open the claw to release the object
10) Return to step 2) and repeat.

The high-level logic outlined above is implemented in `main.m` on the MATLAB side.

[1] $q_0 = 0$, $q_1 = \pi/2$, $q_2 = 0$

III. RESULTS AND DISCUSSION

A. Displaying Applied Force Vectors on a Live Plot

One of the tasks that took the most time was to get the arm to correctly display the force vector, mainly due to us trying to interpret the angle of the arm as radians, when the input was encoder ticks. Although we did eventually fix our Jacobian-based approach, during the time that it was not working we took a a physics approach to computing this vector (see figure 10). For a static system in a 2d plane, two torque equations can be made, one for τ_1 and one for τ_2:

$$\tau_2 = fcos(\theta_3) * l_2$$

$$\tau 1 = (l_0 - z)fsin(90 + \theta_3 + \theta_2 - \theta_1) + xfcos(90 + \theta_3 + \theta_2 - \theta_1)$$

and the two unknowns can be solved. As θ_3 was embedded deep within sin and cos in τ_1, the mathematics to solve for it would be nontrivial. Due to this, the answer was solved through MATLAB. The symbolic `solve()` function, and symbolic `simplify` function was used to solve for the magnitude and angle, displayed in corresponding order as the first, and second answer. While this method was unimplemented, this did show how greatly the Jacobian reduced the amount of mathematics to be implemented by the user.

B. Live Object Measurement

The final output of this step is a set of three graphs (figure 9). The first graph is a position vs time graph of the arm as it moves back and forth in the xy plane. The arm swings from $+y$ to $-y$ while ever increasing in the x, and not changing in the z. The graph plots this exactly as is specified by the code: The y position creates somewhat of a square wave, bouncing back and forth between 3 and -3, the x increases from 0 to 4, and the z stays stable. The starting inconsistency that one can tell, however, is due to the fact that the arm did not start at the first position in the list. There was no trajectory planning for this point, it was only set and assumed to be there initially. Because of this, when the program started to run, the arm moved at high velocity to this start position. The second plot is a Force Vector vs Time graph. Again, as expected, the force in the z direction remained very consistent throughout each of

the tests. This was because the arm did not accelerate in the z direction, causing no extraneous torques or forces on the arm. As the arm moves from close to its center to the max range "out" in the x direction, one can see the force changes from being mainly in the y direction to mostly in the x direction over the course of the movement. Each spike in the x direction is due to the centripetal force of the arm moving across each point, while each spike in the y direction is due to the acceleration of the cubic trajectory, causing a speed from 0 to 0 over the duration. The net force for the arm, as one would expect, remains decently consistent between each loop. Each slight dip in the net force, every 1.5 seconds or so, is explained by the arm settling from motion. This settling stops the acceleration in all directions except gravity, and, at this point, only the mass of the arm and weight is read.

C. Problems in Automated Sorting

Implementing the basic functionality of the arm so that it would pick up the mass, weigh it, and deposit it in the desired location based off of it weight and color, was a process that we were able to get done relatively easily; however, most of our time and energy went into getting the arm consistent for each trial, trial after trial and day after day. There were 3 problems we ran into: inconsistencies in location, discrepancies in force sensing, and malfunctions in gripping.

1) Inconsistencies in Location: We ran into two inconsistencies while trying to track the objects of each color, in reading the objects themselves and in tracking their absolute position. First, the color readings were not consistent. Depending on the ambient lighting and focus on the camera, the colors would be read differently. By having a light source brighter or closer, the color would seem lighter then the calibration hue, preventing the sphere from being read as a solid circle. If the focus was blurred from the center value, the colors would blend together, and the resulting read would be a smaller sphere. When both of these built up to be too great, the spheres could be deteriorated to such an amount that they could not be read at all, or be lower than our threshold size for an object. To fix this, we fixed a lab light exactly 22.5 inches above the center of the board and calibrated the color thresholds for this light source.

Second, the arm was unable to go the the correct location of the colored ball initially. This was because the height of the ball above $z = 0$ caused the camera's angled projection to see the ball on a flat plane further back than its position should be. Originally, the solution to this was to add a set value to the x position, increasing the position the arm would go to forwards single inch. However, as we tested the edge cases, we noticed another inconsistency, the placement closest to the camera would miss the slightest bit, while further back would miss by a relatively great value. We decided to implement a second patch to the y position, as a dependency on x. The end result was 2 equations for our shift of position: $x = x + 0.875$ and $y = y * (a_1 + a_2 x)$, which allowed for great accuracy in correctly grabbing the specified object, regardless of location.

2) Discrepancies in Force Sensing: One can see, when looking at figures 1 - 8, that there is a need to show the unloaded vector to accurately read the data. In addition, were one to test the robot after leaving it over night, they would find something peculiar: it would have a tenancy to sort both heavy weights and light weights as light weights or to sort both as heavy weights. There are two main reasons that a vector is displayed, even when the system is unloaded. First, gravity compensation is not implemented. This means that the force created by the innate weight of the arm is not counteracted, and the vector is the net value, not just the applied value. The second is due to the joint torque sensors themselves. They had a tenancy to drift greatly depending on how long they were used for. Most likely, this was due to changes in temperature of the sensor. When the arm was on for a long time, during 2 days of multiple hour testings, the heating caused by the current running through the joint torque sensors, along with the ambient temperature, caused a shift in the readings, especially when the end effector was held far from the center position. A higher temperature means a higher thermal expansion of the strain gage and a higher lead wire resistance, both of which lower the voltage output by the whetstone bridge. The wires connecting the strain gage to the bridge were extremely tiny, as they were the same thickness as on the surface of the gage itself. On the surface, this is done in the gage so that any flexing of the wire can be sensed, however, outside of this, having the external wires so close to the bot, a wrong movement could push or crimp the wires, simulating, sometimes permanently, a strain on the robot. Both of these cases force recalibration when brought to an extreme, as the data we have used to measure the forces are no longer accurate.

3) Malfunctions in Gripping: When we first acquired the claw and attached it to our robot arm, we

noticed that it was suitable enough to pick up the light objects and move them around without dropping them. However, when we tried to move the arm around with the heavy object in the claw, the object would fall out if the arm moved to fast. Our first thought was to slow down the trajectory generation of the arm from point to point. We tried this and it worked but there were still some runs where the weight would fall out of the claw. We knew that this was not consistent enough so we needed to seek another alternative. We decided that 3D printing an attachment in the shape of the spherical object would be our best chance at the robot not dropping the heavy object. After designing and 3D printing our first iteration of the design, it proved to have an error in the dimensioning and was too small to encase the entire object.

Instead of 3D printing the first design again with the correct dimensions, we scratched it and moved onto a forklift method. After designing and 3D printing this design, we tested it and it was better than the claw alone because it did not allow the object to rotate when going up to the weighing position. While it was better in this sense, it was not suitable enough to hold onto the object when the arm was in a downward position thus, was not an acceptable solution.

After many iterations, we decided to make the claw have more grip instead of trying to make more attachments for it. Wrapping high-friction rubber electrical tape around each part of the claw proved to be the best and easiest way to hold onto the heavy object. This was our final iteration of the claw. After the claw's grip was improved, we were able to double the speed of our trajectories without any negative side effects to the reliability of the system.

IV. CONCLUSION

Overall, this last project was a very good way of utilizing what we have learned in all of the previous labs and in the class in general. It brought all of the components together, including the forward and inverse kinematics of the arm, force sensing using the load cells, and vision processing using the camera. One of the main underlying issues that we experienced was that the joint torque sensors drifted. To correct this, we needed to calibrate them for each session that we were working on the robot. While this was not much of a hassle, it did prove to hinder the program if not calibrated correctly. Another method that needed to be explored for our arm was the claw. Our gripper was not suitable enough to grip the heavy object while the arm was in motion. To correct this, we stepped through multiple iterations of the claw and finally came to a conclusion of putting high-friction rubber electrical tape around each portion of the claw. While we did have some barriers to overcome, we were able to successfully finish the final project and have the robot reliably sort the objects by both color and weight.

The code for this project can be found on Github in our team's Matlab and Nucleo firmware repositories under the tag `rbe3001_final`.

ACKNOWLEDGMENT

We would like to thank professor Fisher and Kevin Harrington for providing us with the materials and knowledge necessary to complete this project. We would also like to give special thanks to our teaching assistants, Gunnar Horve and Nathaniel Goldfarb.

APPENDIX

The figures below represent the arm in three different measurement positions to measure the 5N force that we exerted on the arm. There are two figures per configuration, one representing no load on the arm and one representing the 5N force on the arm. As you can see in each figure, we have shown the force that is being applied to the arm using the load cells. The next three numbers in the figure represent the torque in each joint of the arm going from q0 to q2 respectively.

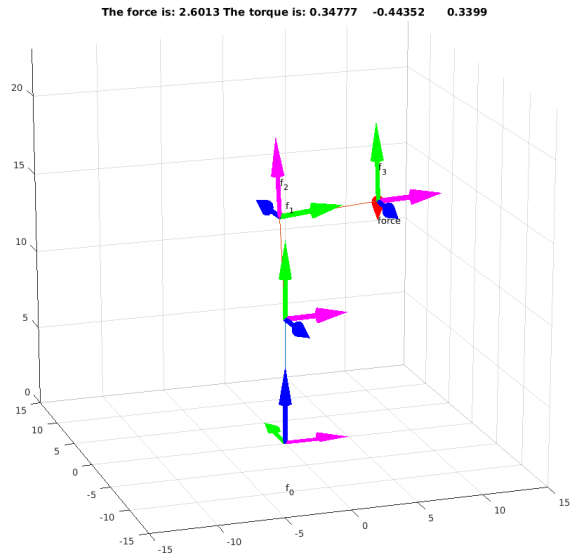

Fig. 1: Example of first position with no load

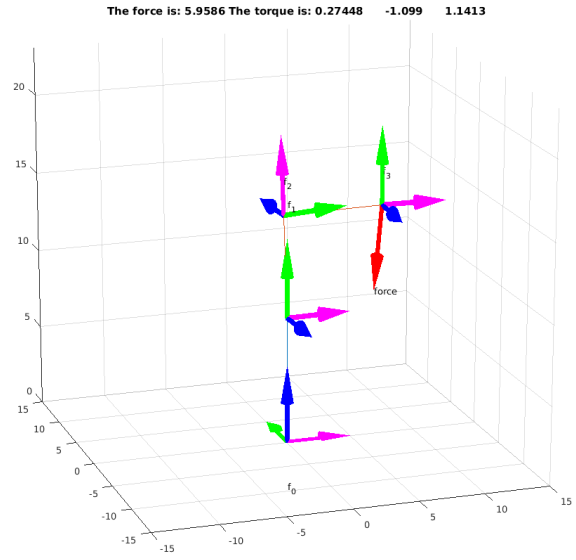

Fig. 2: Example of first position with load applied

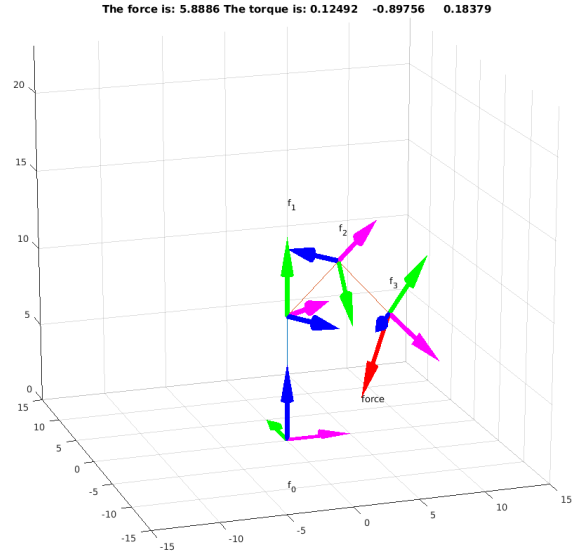

Fig. 3: Example of second position with no load

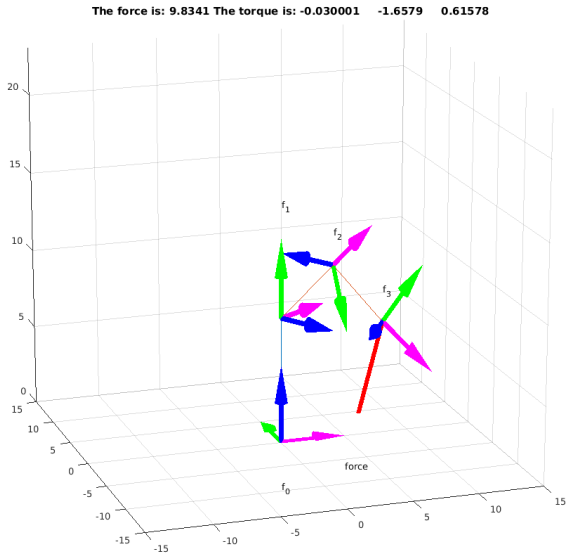

Fig. 4: Example of second position with load applied

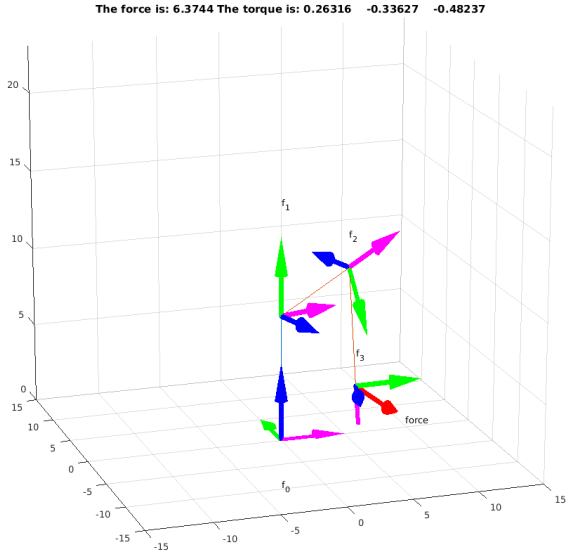

Fig. 5: Example of third position with no load

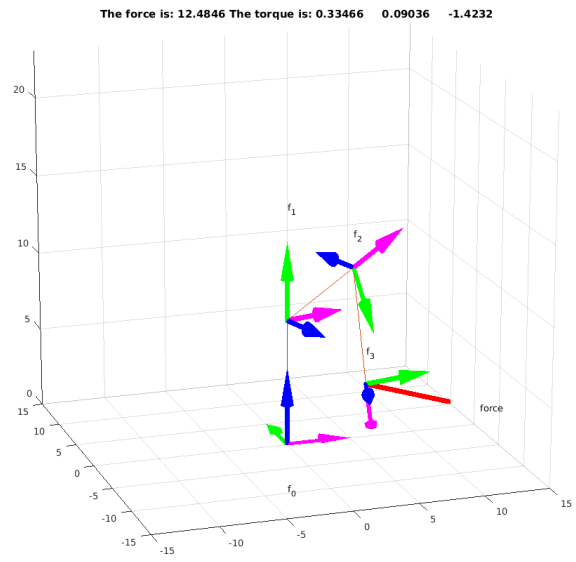

Fig. 6: Example of third position with load applied

The next two figures below represent the arm in two different configurations while holding the object. For the second figure, our calibrations were off for the initial starting value, but the end value ended up being 20N when it started at about 18N. The force that is being represented is the object and the object weighed about 4N.

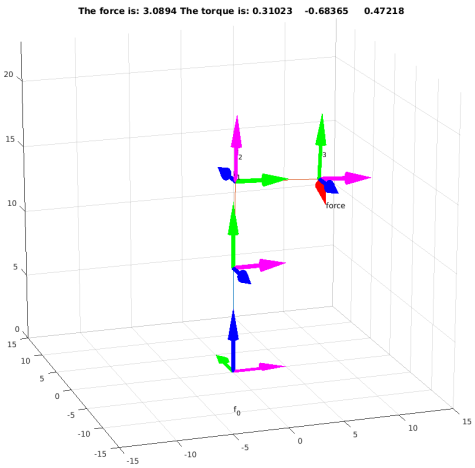

Fig. 7: Example of first position with object

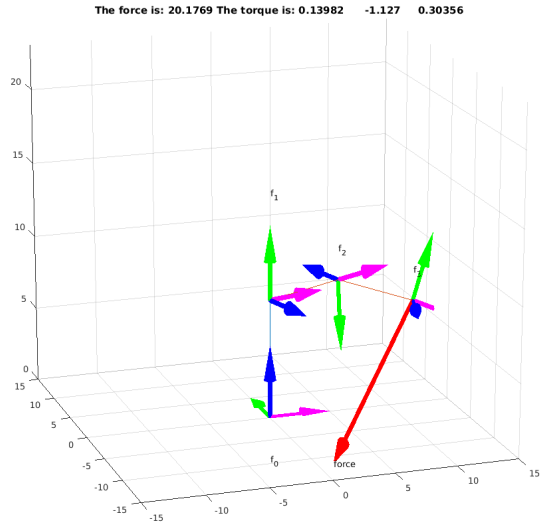

Fig. 8: Example of second position with object

For this part of the project, we needed to make the arm go through a series of 10 arbitrary points while holding the object in the gripper, stopping at each setpoint for at least five seconds. For the plot below, this describes the x, y, z position, the force, and the Net force all with respect to time.

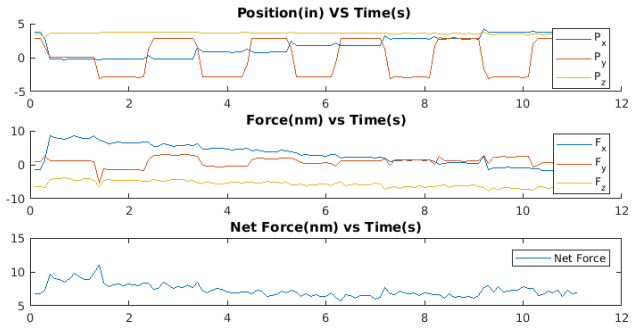

Fig. 9: Plot of Live Object Measurement

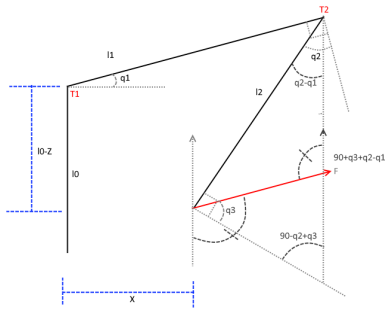

Fig. 10: A Physics Approach to the Torque of System

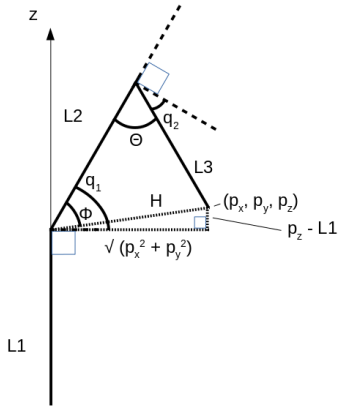

Fig. 11: Diagram for Inverse Position Kinematics

$$\begin{bmatrix} -L2 * sin(q0) * cos(q1) - L3 * sin(q0) * sin(q1 - q2) & -L2 * cos(q0) * sin(q1) + L3 * cos(q0) * cos(q1 - q2) & -L3 * cos(q0) * cos(q1 - q2) \\ L2 * cos(q0) * cos(q1) + L3 * cos(q0) * sin(q1 - q2) & -L2 * sin(q0) * sin(q1) + L3 * sin(q0) * cos(q1 - q2) & -L3 * sin(q0) * cos(q1 - q2) \\ 0 & L2 * cos(q1) + L3 * sin(q1 - q2) & -L3 * sin(q1 - q2)]; \end{bmatrix}$$

Fig. 12: The 3 by 3 truncated Jacobian calculated for the 3DOF arm

REFERENCES

[1] http://www.ni.com/white-paper/3432/en/
http://www.sciencedirect.com/science/article/pii/S2238785414001082
https://www.cirris.com/learning-center/general-testing/special-topics/177-temperature-coefficient-of-copper
https://www.mathworks.com/matlabcentral/fileexchange/25372-marrow3-m-easy-to-use-3d-arrow