RBE 3001 Lab 5 Final Project

Nick Benoit, Mary Hatfalvi, and Daniel Wivagg

***Abstract*— The final project for RBE 3001 combined forward and inverse position and velocity kinematics, dynamics, and vision processing for the control of a 3-DOF robotic arm. The manipulator was able to pick up objects in a 3-dimensional workspace and sort them by weight and color. A combination of MatLab programming skills, robotic control theory, and C++ programming knowledge was required to complete the project.**

I. INTRODUCTION

This project was a culmination of work done with kinematics, force sensing, and manipulation using a 3-DOF robotic manipulator. The goal was to sort objects in the workspace by color and weight using strain gauges and computer vision. First, the gauges were calibrated to return accurate values of the torque at each joint of the arm based on the unique properties of the arm and a provided scaling factor. Then, the camera was configured to search for three colors of objects in the workspace based on their size and Hue, Saturation, and Value (HSV) parameters. In order to pick up objects, a gripper was attached to the end of the manipulator and programmed to open or close based on a command sent to the micro-controller's firmware. Finally, using a state machine and trajectory generation from previous laboratory exercises, the arm picked up and sorted objects based on the color and weight information gathered during the sorting process.

II. METHODS

A. D-H Parameters and Position Kinematics

Forward position kinematics are used to find the arm's (x,y,z) position in task space from the joint parameters. Forward position kinematics are derived from frame transformations using D-H parameters. For the 3-DOF arm, configured as shown to the right in Figure 1, the D-H parameters are written as follows:

Link	α	a	d	θ
1	90 deg	0	l_1	$q_0(t)^*$
2	0	l_2	0	$q_1(t)^*$
3	0	l_3	0	$q_2(t)^*$
4	0	0	0	90 deg

The generalized transformation matrix is a 4x4 matrix with rotation matrix R position vector P.

$$T_a^b = \begin{bmatrix} R & P \\ 0 & 1 \end{bmatrix}$$

In this case, the rotation matrix is defined:

$$R = \begin{bmatrix} \cos q(t)^* & -\cos\alpha \sin q(t)^* & \sin\alpha \sin q(t)^* \\ \sin q(t)^* & \cos\alpha \cos q(t)^* & -\sin\alpha \cos q(t)^* \\ 0 & \cos\alpha & \sin\alpha \end{bmatrix}$$

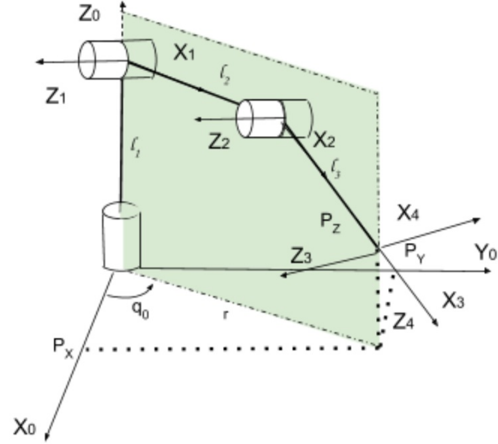

Fig. 1. 3-Dimensional view of the manipulator

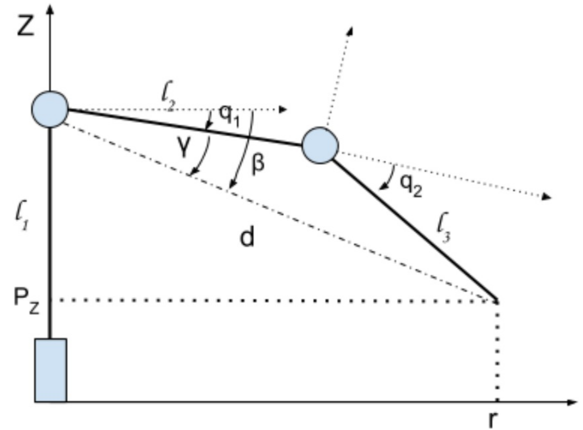

Fig. 2. Side profile view of the manipulator

The position vector can be defined:

$$P = \begin{bmatrix} a\cos(q(t)) \\ a\sin(q(t)) \\ d \end{bmatrix}$$

The four transformation matrices are found by evaluating T_a^b with the values for each α, θ, a and d according to the D-H parameters.

$$T_0^1 = \begin{bmatrix} \cos(q_0(t)^*) & 0 & \sin(q_0(t)^*) & 0 \\ \sin(q_0(t)^*) & 0 & -\cos(q_0(t)^*) & 0 \\ 0 & 1 & 0 & l_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_1^2 = \begin{bmatrix} \cos(q_1(t)^*) & -\sin(q_1(t)^*) & 0 & l_2\cos(q_1(t)^*) \\ \sin(q_1(t)^*) & \cos(q_1(t)^*) & 0 & l_2\sin(q_1(t)^*) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_2^3 = \begin{bmatrix} \cos(q_2(t)^*) & -\sin(q_2(t)^*) & 0 & l_3\cos(q_2(t)^*) \\ \sin(q_2(t)^*) & \cos(q_2(t)^*) & 0 & l_3\sin(q_2(t)^*) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_3^4 = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The full forward position kinematics is the product of each link transformation matrix.

$$T_0^4 = T_0^1 * T_1^2 * T_2^3 * T_3^4$$

The symbolic transformation matrix was calculated in MatLab. It was used to model the arm in real time as well as to find the manipulator Jacobian matrix. The rotation and position components of this matrix are shown below.

$$R = \begin{bmatrix} \cos(q_2+q_3)\cos q_1 & -\sin(q_2+q_3)\cos q_1 & \sin q_1 \\ \cos(q_2+q_3)\sin q_1 & -\sin(q_2+q_3)\sin q_1 & -\cos q_1 \\ \sin(q_2+q_3) & \cos(q_2+q_3) & 0 \end{bmatrix}$$

$$P = \begin{bmatrix} \cos q_1(l_3\cos(q_2+q_3)+l_2\cos q_2 \\ \sin q_1(l_3\cos(q_2+q_3)+l_2\cos q_2 \\ l_1 + l_3\sin(q_2+q_3)+l_2\sin q_2 \end{bmatrix}$$

To find the inverse position kinematics, links 2 and 3 of the arm were analyzed in a moving plane as shown Figures 1 and 2. The angular position of this plane (shown in green) was determined by the position of link 1. To determine all of the joint values from task space coordinates, the radius had to be calculated first. This value is the vector from the origin to the desired (x,y) position of the tip.

$$r = \sqrt{P_x^2 + P_y^2}$$

The angular position of the plane, or of the first joint, was calculated based on the x- and y-coordinates of the task-space position.

$$q_0 = \arctan\frac{P_x}{P_y}$$

The β and γ values in Figure 2 were necessary for defining q_1, which usually could be expressed as one of two different values based on whether the elbow-up or elbow-down configuration was desired.

$$\beta = \arctan\frac{r}{l_1 - P_z}$$

$$\gamma = \arccos\frac{(l_1-P_z)^2 + r^2 + l_3^2 - l_2^2}{2l_2\sqrt{(l_1-P_z)^2+r^2}}$$

With these values calculated, links two and three could be treated as a two degree of freedom arm. The angle of the elbow joint was calculated first, and then the angle of the shoulder was calculated.

$$q_1 = \beta \pm \gamma$$

$$q_2 = \arccos\frac{(l_1-P_z)^2 + r^2 + l_3^2 - l_2^2}{2l_2l_3}$$

On top of using these equations, the MatLab code required failsafes to prevent the arm from being set to an impossible configuration. To guard against this, try-catch statements in MatLab were utilized to prevent the function from being used with values that were outside the robots physical boundaries of motion. Also, if imaginary values were returned, an error was thrown to prevent the rest of the program from terminating.

Finally, the algorithm decided between elbow-up and elbow-down configurations by assessing the positions of the other links, which were calculated first. In some cases, elbow down would have resulted in an impossible setpoint for the second link, due to the base and ground beneath the robot. Setpoints outside of the robots workspace and trajectories that would collide with the camera stand were avoided using try-catch statements in MatLab.

B. Manipulator Jacobian and Velocity Kinematics

One method of navigating to coordinates in task space without using inverse position kinematics is through inverse velocity kinematics. In this method, the robot is given a direction to move in at a desired speed, rather than a stream of setpoints. The forward and inverse velocity kinematics are found through the 3x1 manipulator Jacobian matrix.

$$\dot{\vec{x}} = J(q)\dot{\vec{q}}$$
$$\dot{\vec{q}} = J(q)^{-1}\dot{\vec{x}}$$

The Jacobian was calculated symbolically through MatLab, and can be expressed through the following set of matrices.

$$J(q) = \begin{bmatrix} J_1 & J_2 & J_3 \end{bmatrix}$$

$$J_1 = \begin{bmatrix} -\sin q_1(l_3\cos(q_2+q_3)+l_2\cos q_2) \\ \cos q_1(l_3\cos(q_2+q_3)+l_2\cos q_2) \\ 0 \end{bmatrix}$$

$$J_2 = \begin{bmatrix} -\cos q_1(l_3\sin(q_2+q_3)+l_2\sin q_2) \\ -\sin q_1(l_3\sin(q_2+q_3)+l_2\sin q_2) \\ l_3\cos(q_2+q_3)+l_2\cos q_2 \end{bmatrix}$$

$$J_3 = \begin{bmatrix} -l_3\sin(q_2+q_3)\cos q_1 \\ -l_3\sin(q_2+q_3)\sin q_1 \\ l_3\cos(q_2+q_3) \end{bmatrix}$$

Although velocity kinematics can be useful for avoiding more complex and geometrically confusing calculations, they were not necessary to complete this lab because inverse position kinematics were simply more effective. However, the manipulator Jacobian was still required for other calculations such as the task-space force calculation.

C. Force Vector Calculation

In order to convert from a set of torques in joint space to a task space velocity vector, the manipulator Jacobian found above was used. Using a strain gauge located at each joint, the force applied at the tip of the manipulator could be determined. The force vector is found using the transpose of the Jacobian in the equation shown below. To convert from joint space to task space force, the inverse of the transpose is required.

$$\vec{\tau} = J(q)^T \vec{F}_{tip}$$

$$\vec{F}_{tip} = (J(q)^T)^{-1} \vec{\tau}$$

This equation was implemented in a MatLab function to calculate the force at the tip based on the readings from the strain gauges. These readings were smoothed with a rolling average in the robot's firmware to ensure a steady and accurate tip force vector.

D. Trajectory Planning

The trajectory planning was calculated through the cubic polynomial equation given as:

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3$$
$$\dot{q}(t) = a_1 + 2a_2 t + 3a_3 t^2$$
$$\ddot{q}(t) = 2a_2 + 6a_3 t$$

The velocity equation is the derivative of the position equation and the acceleration equation is the derivative of the velocity equation. The constants a_0, a_1, a_2,and a_3 are found through the matrix equation with the following constraints:

- t_0 is the start time
- t_f is the final time
- q_0 is the start position
- q_f is the final position
- v_0 is the start velocity
- v_f is the final velocity

The trajectory equations can be expressed in matrix form using these parameters like so:

$$\begin{bmatrix} 1 & t_0 & t_0^2 & t_0^3 \\ 0 & 1 & 2t_0 & 3t_0^2 \\ 1 & t_f & t_f^2 & t_f^3 \\ 0 & 1 & 2t_f & 3t_f^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} q_0 \\ v_0 \\ q_f \\ v_f \end{bmatrix}$$

This matrix was implemented within a function in MatLab to create a trajectory for the x, y, and z directions of motion. The function used the equation for $q(t)$ to generate setpoints for a path with a desired time step between each point. These setpoints passed through the desired target points, so the arm traveled through each one in order to follow the trajectory.

E. Object Detection

The camera was used to detect a dynamic number of objects at random positions in the robot's workspace. To detect an object, the camera would capture a still image at the beginning of each sorting operation, as shown in figure 3. Using the Color Thresholder app in MatLab, a set of three masks were created for blue, green, and yellow objects. Each mask was tailored to the hue, saturation, and value of

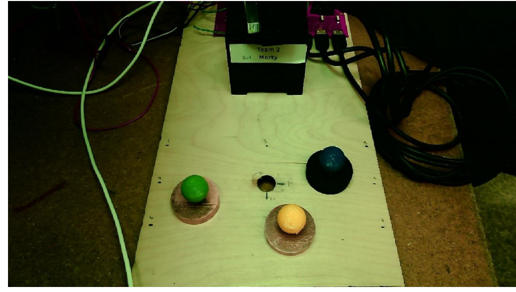

Fig. 3. Still image of robot's workspace

one object. By applying these masks in succession, a binary image of each color was created for the workspace. Due to the similarity between the color of the sorting objects and objects near the robot's workspace two filters were employed to reduce the likelihood of a false positive. A size filter was used to remove detected objects with less than 60 pixels, and an area filter was used to crop off regions of the image outside the workspace. This reduced the noise in the binary images. The centroid for objects of each color were calculated separately using the cleaned binary images for each color, as shown in Figure 4. The location of the

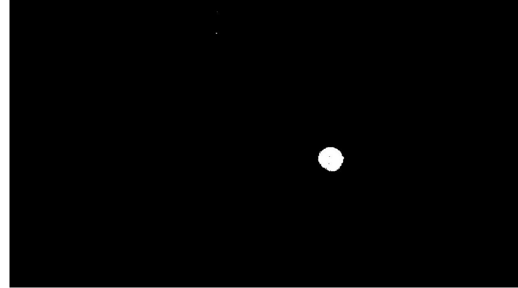

Fig. 4. Binary image created using blue mask

centroids in pixels was converted to workspace coordinates.

F. Sorting Plan

In order to sort the objects consistently, the robot was configured to sort according to a pre-determined pattern. As shown in Figure 5, the robot placed heavy objects to the right of the workspace and light objects to the left. Blue objects were placed at the top, green in the middle, and yellow at the bottom.

The sorting process consisted of four distinct steps:

1) Locate objects in the workspace
2) Grab one object (first blue, then green, then yellow)
3) Weigh the object at the weighing configuration
4) Sort the object as shown in Figure 5

These steps are represented in the state machine used to control the flow of the sorting program, shown in Figure 6.

III. Results

The implementation of this lab required us to successfully synthesize and modify portions of previous labs, as well as

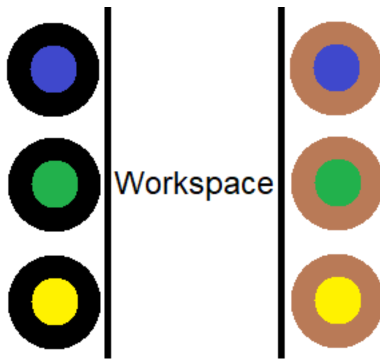

Fig. 5. Method of sorting objects by weight and color

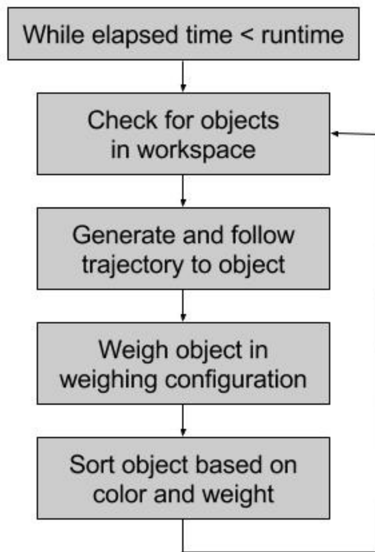

Fig. 6. State Machine process

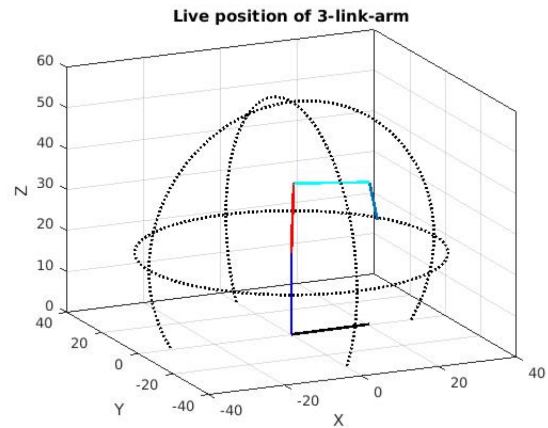

Fig. 7. 3-D Plot of 2N on tip of arm

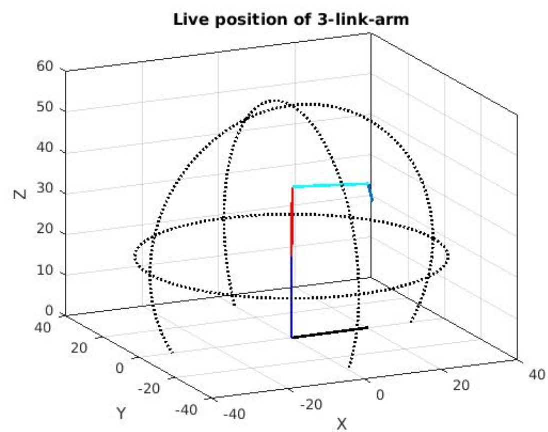

Fig. 8. 3-D Plot of no force on tip of arm

create entirely new functions. Building off of our work in lab 4, we created functions to weigh the workspace objects. Displaying the force vector in our live model was vexing. The method we used to average the force readings caused the displayed vector to lag. We also struggled to display the vector in the correct coordinate frame, but ultimately we were able to successfully implement a force vector in the live model.

The figures below show the force vector on a live 3-D plot of the arm. The live plot shows the robot's workspace range with dashed circles, and the links of the arm are colored blue, red, and cyan. The force vector is drawn from the tip of the third link in the direction of the force, with its length representing the magnitude of the force. Figure 7 shows the force vector measured with a 2N weight gripped by the robot. Figure 8 shows the force vector measured with no weight at the tip. The force vector from the arm with the 2N mass is larger than the force vector from the arm without a mass.

After implementing force reading capability, we logged the positions and forces in each direction as the robot passed through 10 different setpoints holding a heavy weight. The position graphs depicted the motions of the links and the slope of these lines show the velocities. The forces roughly corresponded with the motion of the robot. We also recorded the magnitude of the force applied to the tip, which was relatively constant with some spikes in the data as the robot made quicker motions. The data collected is included in the Appendix and shown in Figure 12 of this report.

Our functions for determining the color and location of objects in the workspace were modified from lab 3, and they performed well. The robot accurately detected the color of objects during each sorting operation. The robot struggled, however, to find the exact location of the centroid. This was due to the angle of the camera in relation to the workspace, causing the robot to consistently miss the objects by about 2 centimeters in the Z direction.

Figure 3 above depicts the workspace from the camera view with one blue object, one yellow object, and one green object. The image is processed using one mask for each color and two filters to create clean binary images. The masks remove pixels that are not within the specified hue, saturation, and value range. A size filter removed objects of less than 60 pixels to reduce noise, and an area filter removed

all pixels outside of the designated workspace. These two filters removed noise from the binary images and reduced the chance of reporting the centroid of a non-object.

The figures below show binary images for each color object: blue, green, and yellow. The binary images only contain pixels with value of 0 or 1. By converting the image from full color to binary, it is easier for the program to distinguish objects from their environment. Figure 9 shows the binary image created using the blue mask. The only pixels with a value of 1 are those that had HSV values within our defined tolerances and were within the robot's workspace. Figure 10 and Figure 11 below show sample binary images created using the green and yellow color masks, respectively.

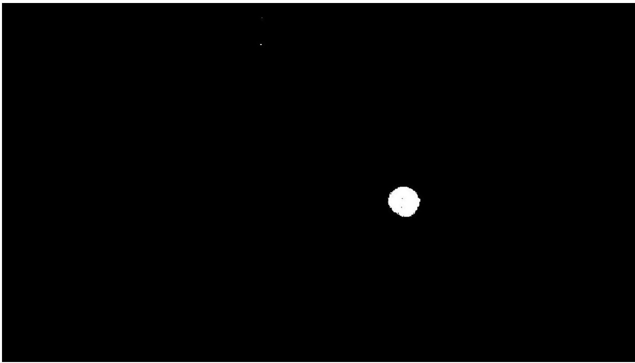

Fig. 9. Binary image created using blue mask

Fig. 10. Binary image created using green mask

From these figures it is easy to see how the color masks and filters turn complicated, color images into simple binary images that can be used to determine centroids of objects.

The robot also encountered issues while moving to pick up the objects and frequently stalled very close to the objects. This was caused by our use of static PID gains to control the robot's motion. The robot was able to pick up and hold objects without dropping them every sorting operation, and placed the objects in the correct area with 100 percent accuracy.

IV. Discussion

The most lengthy and challenging portion of the project was correctly displaying the force vector. Although the manipulator Jacobian and associated parameters were calculated correctly, the output was consistently wrong. Hand calculations and tests with the forward Jacobian did not reveal an obvious cause for the errors. Ultimately, we determined that the frame rotation at the tip was not accounted for and the force vector simply needed to be rotated to be correct.

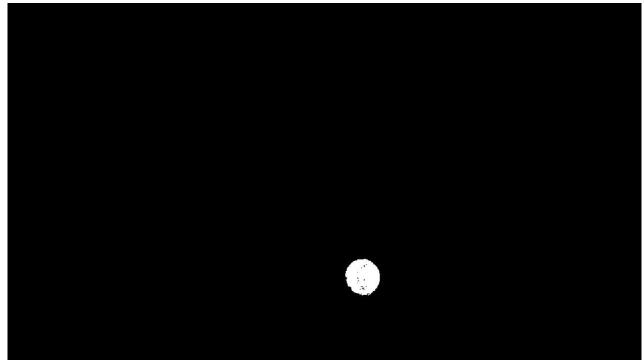

Fig. 11. Binary image created using yellow mask

Another difficulty was found in correctly calibrating the camera because it was replaced between lab 3 and lab 5 with a broken, green-aliasing camera. It was not enough to carefully tune the HSV parameters to detect blue, green, and yellow objects. The only way to mitigate camera issues was to situate a bright lamp above the workspace so that colors could be distinguished correctly. The lamp also created a uniform bright light which was more consistent than the lighting in the environment.

These setbacks were surpassed and the robot successfully identified and sorted objects, but the process was not perfect. At some points during a trajectory, the robot stalled because the PID gains were not high enough to move it out of its position. This is because the general gains were only set once at the beginning of the program, and were suboptimal for certain configurations where the robot required greater gains. If we had increased the gains for the entire program, the system would have become unstable. Using the PID setter command we developed for the packet processor, we could have updated them dynamically for different robot positions. If there was more time, this would have been the ideal way to program the robot. Instead, we compensated for any errors by manually assisting the robot if it got stuck.

Another process imperfection was the slight error in the camera coordinates during image processing. Since the camera viewed the workspace from an angle instead of directly overhead, the centroids returned were not a perfect reflection of the object's task space position. If there was more time, some tweaking of the camera calculations and (x,y) coordinates would have made the process more accurate. The temporary solution used was to slide the objects a centimeter or two so they were positioned directly under the gripper.

V. CONCLUSIONS

All of the essential components of this lab were completed successfully. The robot was able to pick up and sort a

blue, yellow, or green colored object using computer vision, inverse kinematics, force sensing and trajectory planning. Future tasks to improve the functionality would include programming a more accurate object identification algorithm, dynamically updating PID gains, and implementing gravity compensation. Using a technique such as Kalman filtering or another similar algorithm would make the force sensing more accurate. However, given the time limitations of this project, it was largely effective at its tasks. The combination of hardware, electrical, and programming skills required presented a reasonable challenge that mimicked real-world robotics in an interesting way. The introduction to trajectory planning, robot kinematics, and computer vision will undoubtedly be useful in future classes and projects.

APPENDIX

The charts shown in Figure 12 below depict the position, force, and magnitude of force applied at the tip as the manipulator moves through 10 different setpoints.

ACKNOWLEDGMENT

We would like to thank the RBE 3001 course staff for their help with completing this project and debugging hardware and software issues. We would also like to acknowledge Professor Greg Fischer for his role in teaching the fundamental mathematics required to program the manipulator.

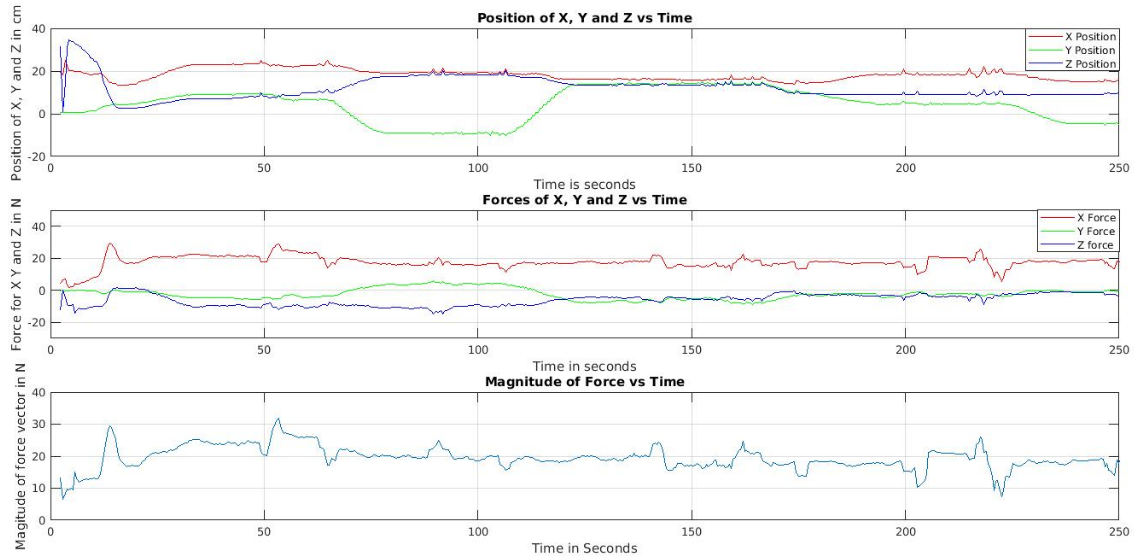

Fig. 12. Position, Force, and Magnitude through 10 setpoints