Final Project: Robot Pick and Place System

Team 17: Matthew Gomes, Omri Green, and Grace Holden | *March 2, 2022*

Abstract— **In this lab, we implemented a robotic pick-and-place system using techniques developed in previous labs, as well as an object detection algorithm based on color masking. Our system was designed to be dynamic, responding seamlessly to environmental changes such as the removal or addition of objects of interest, and as such the program architecture was designed to be efficient and non-blocking, offloading computationally expensive image processing to a separate CPU core from our main state machines. Our image detection and kinematic calculations integrated relatively seamlessly, allowing us to create a robust and organized system which accomplishes the desired goals accurately and consistently.**

I. INTRODUCTION

For the robotic pick and place system we combine and build from what we have learned in previous labs to make a sorting system with our three degree of freedom robotic arm. We incorporate computer vision to identify objects, localize them, move the robotic arm, grab the objects, and sort them. We utilize our knowledge functions from previous labs to control the motion of our arm, while introducing a focus on new topics in computer vision to identify, classify, and sort objects on the board based on color and shape. We work through these tasks using a color masking approach to segment and localize objects of varying colors and shapes.

II. BACKGROUND

In this project we will combine what we have learned in the previous labs. In the first lab we learned about the system architecture of the Hephaestus robot arm and wrote MATLAB code to send commands to the joints of the robot. Beside sending commands to the arm, we also monitored the position in joint space.

Building off of these principles, in the second lab we calculated the forward kinematics of the arm in order to determine the position and orientation of the end effector. We also worked on visualizing the arm and its position in MATLAB using stick diagrams which could be verified by manually moving the arm.

The third lab involved calculating the inverse kinematics of 3-degrees of freedom arm and implementing those calculations in MATLAB. We created a trajectory generator function and considered trajectory planning in joint-space, as well as in task-space. This functionality allows us to transition more smoothly from one point to another.

Eventually, in the fourth lab we worked with differential, or velocity, kinematics. We did this by calculating the Jacobian matrix and therefore the forward velocity kinematics of the robot. We wrote a program in MATLAB to implement these calculations for the robot. We also implemented a numerical approach to solving an inverse kinematics problem. Finally, we performed trajectory following using speed and direction commands, followed by visualization and characterization of the motion trajectories of our arm. In the end we had implemented velocity-based trajectory following, an iterative inverse kinematics solution, and a live 3D CAD model of the arm.

III. MOTIVATION

This lab allows us to integrate the topics we have been working on over the course of the term into a final cohesive project. Incorporating all of these topics allows us to solidify our understanding of robotic systems and object recognition using the camera attached to the board. Throughout the term we have worked on finding the best way to move the arm to desired positions, tracking those positions, and manipulating our trajectories to avoid errors caused by singular configurations. In this final project we put these individual components to use in a tangible way. After this lab the arm will be able to recognize, classify, move, and sort objects on the board. While this project is implemented on the relatively small scale of the robot checkerboard, the skills are relevant to much larger applications beyond or within our future robotics coursework. It is also useful to have these tangible instances of the topics we are learning about in lecture as they can only help us further our understanding of robotics.

IV. METHODS

A. Camera Setup & Intrinsic Calibration

First, we found and set up the camera in MATLAB using the camera support package. The camera is very sensitive to lighting in general. Shadows and differences in lighting due to natural light in the lab can make it difficult to calibrate the camera to the board and recognize objects. In order to mitigate the effects of inconsistent lighting, we used a bright light overhead and positioned it slightly to the right side of the board (from the camera's perspective) which typically displayed as poorly lit in our initial photos. For the intrinsic calibration of the camera we used the camera calibration app in MATLAB. We took numerous photos of the checkerboard in order to get a calibration where the MATLAB was able to see the points on the board and register the entire workspace. We had to add a strip of painters tape along the back row of the board by the base of the arm because we did not want that portion of the board to be included in our calibration and in our initial calibrations, it was being registered. We took about 80 images in order to get 20 that would work. Before we could run the calibration, we made sure that our X and Y axes were aligned consistently and correctly in all the photos.

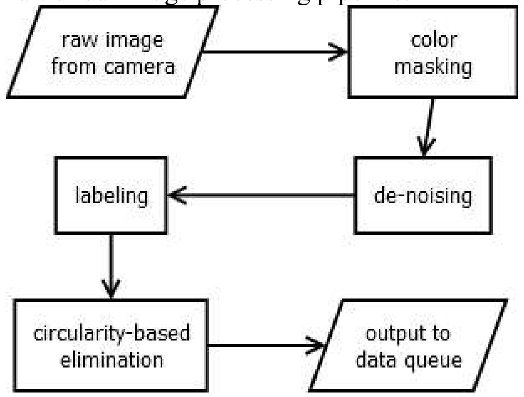
To run the calibration, we had to select the camera model and make sure the radio button indicating the camera model "fisheye" was selected before pressing the "Calibrate" button. Then we got rid of images that were not helpful or had an error that was not acceptable. Once we were happy with the calibration results, we saved them by clicking "Export Camera Parameters" which allowed us to generate a script to be readily available every time we come into the lab so that we would not have to continue to recalibrate the camera every time we worked on our project.

B. *Camera-robot registration (Extrinsic Calibration)*

We registered the camera between the reference frame of the robot and the reference frame of the image (Pixel Coordinate in the Image Plane). This relates the position of objects within the field of view of the camera to robot task space coordinates. The function getCameraPose() calculated the transformation between the robot's base frame and the checkerboard's frame. We obtained a position of interest in x-y pixels in the image frame using the data tips tool. Then we used the pointsToWorld() MATLAB function to transform the image frame to a checkerboard frame. This is a built-in function from MATLAB's image processing toolbox which takes in the intrinsic calibration parameters, the extrinsic calibration parameters, and the (x,y) pixel coordinates in the image and returns the corresponding x,y location in mm with respect to the grid-attached frame on the paper. We then found the transformation matrix from the robot's base to the checkerboard's frame. Finally, we multiplied the transformation matrix by the pose in the checkerboard to get the position of the target we identified in pixel coordinates in the robot's base frame coordinates. We then validated the correctness of our camera-robot calibration registration.

C. *Object Detection and Classification*

Our image processing pipeline was implemented using a color masking approach in a MATLAB parallel pool. As our cameras were not compatible with MATLAB parallel pools (reading from them gave nothing but timeouts and hardware inaccessible errors), we had to stream our camera's input through OBS's virtual camera feature, and then access that feed from MATLAB. Frame acquisition was implemented differently from how it was in the provided sample code- we discovered that starting the camera as a video feed and then simply pulling frames from that feed as needed while clearing the video buffer to prevent out-of-memory errors gave the lowest-latency response. After frame acquisition, we created four masked variants of the image using color masks tuned to separate out red, green, yellow, and orange objects into black/white images. We then ran our object detection algorithm on each color variant independently. This algorithm consisted of the following steps: (1) we removed any included segments under a certain area to filter out "noise" (small areas unintentionally included by the mask) using bwareaopen(), (2) we label the remaining regions using bwlabel() , (3) we retrieved the area, perimeter, centroid, convex hull (smallest polygon that contains a region) and the major and minor axes (axes of the smallest ellipse which contains a region) of each remaining region using regionprops(), and finally (3) we use the gathered information to compute (a) the circularity of each region (perimeter^2/(4*pi*area)), and (b) for each area of adequate circularity we compute a corrected centroid by finding the midpoint of the longest cross-section of the convex hull. These remaining corrected centroids are reported to the main state-machine thread as likely locations for objects of the relevant color after being converted to the task-space using pointsToWorld() with our camera calibration, as well as a simple rotation and translation. We experimented with using the major and minor axes to detect overlaps between objects of the same color, but although the calculation was functional, the end-effector of our robot was not dexterous enough to accurately separate balls that close together, so the functionality was disabled. The following is a flowchart of our image processing pipeline:



D. *Object Localization*

In this step, we convert the centroid location of the balls into usable target positions for the robot to pick up. We used our previous work to determine the target centroid in the image and then related that centroid to a point in the plane of the checkerboard using . Next, we had to account for the fact that the balls are not flat circles lying on the grid. Simply using the centroid of a detected 2d circle will not account for this in most orientations, so we provided the robot with an offset for centroid locations to account for this effect.

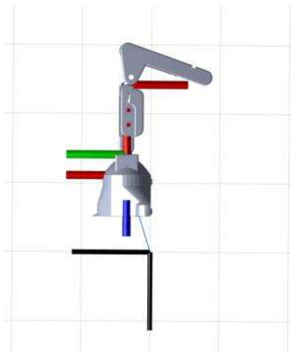
E. *Final Project Challenge*

Here we use what we have developed in the previous steps in order to implement a program that:

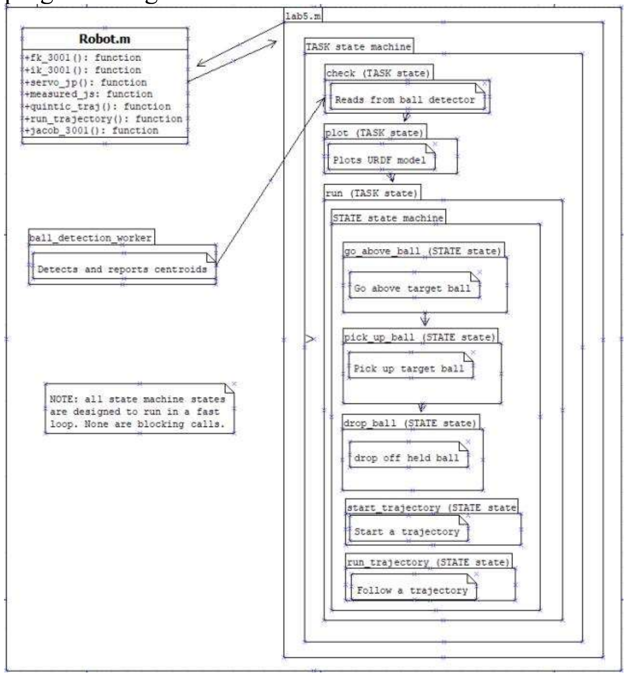
a.) determines the 2d position of an object with respect to the robot's reference frame
b.) Uses inverse kinematics from previous labs to calculate the joint angles required for the robot's end effector to reach the object's location.
c.) Uses trajectory planning to implement a smooth motion from the robot's current position to the location of the object. (We moved to a position above the object and then moved slowly down towards the object to avoid hitting and displacing the orb.)
d.) Close the gripper. The function "closeGripper" was already written and provided in the Robot class.
e.) Pick the object up, move it to an arbitrary position, and drop it by opening the gripper. The function

"openGripper" was already written and provided in the Robot class.

We integrated everything we have done in the labs to configure the system to detect relevant objects in the camera's field of view. We worked to sort the objects based on color, lift them with the arm, and distribute them to pre-determined locations (pick-and-place). Our pick-and-place system was designed to handle arbitrary amounts of every color of object, and was also dynamic- that is, it could account and correct for changes in object location during its trajectory. We also demonstrated its ability to detect, categorize, and grab an object other than the lab balls (a marker cap, in our case), and implemented a dynamic URDF render of our robot to visualize it in 3d space throughout its tasks, shown below:



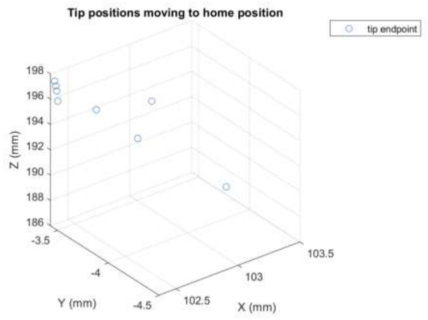
Our pick-and-place program was modeled as two nested state machines, TASK and STATE. The TASK machine cycled through three states: check, plot, and run, for each task of checking for changes, plotting the URDF, and running the robot, respectively. The STATE machine controlled the current goal of robot movement in the pick-and-place workflow. Simultaneously, a worker thread ran on another CPU core to keep the TASK machine fed with up-to-date centroid locations. Below is a diagram of the program design:



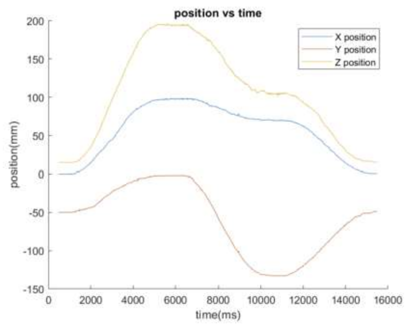
V. Results

Our robot was able to accurately and consistently complete all assigned tasks, dynamically tracking and sorting arbitrary numbers of balls of all colors without issue. Data gathered regarding the precision of several components of our code are listed below:

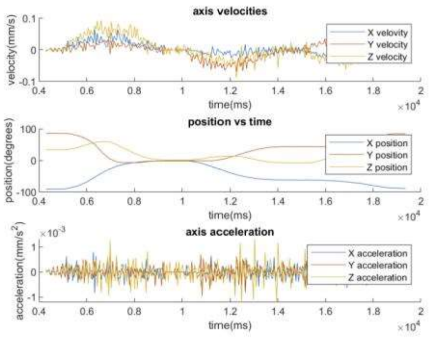
Using forward kinematics to calculate the robot end-effector position while moving repeatedly between the nest and zero positions, we measured an average real-world tip end position of (102.5, -3.8, 195.9) in the zero position, compared to the optimal (100, 0, 195). We calculated a root-mean-square error of 5.502mm.



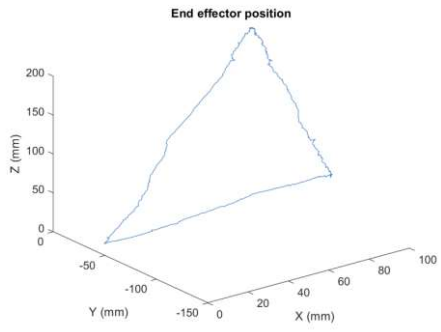
Using inverse kinematics and quintic task-space trajectory generation, we moved our robot between three locations in the X,Y,Z plane to visualize the accuracy of our model. As shown by graphs of the X,Y,Z position of the end-effector over time throughout the motion,



Velocity, acceleration, and joint position over the same,

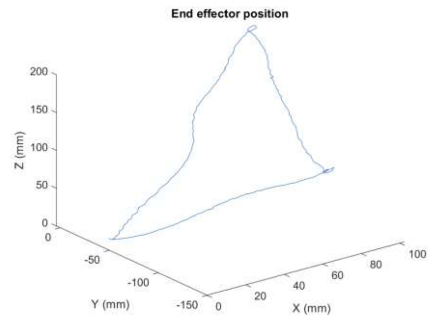


And the 3D end-effector position throughout the trajectory,

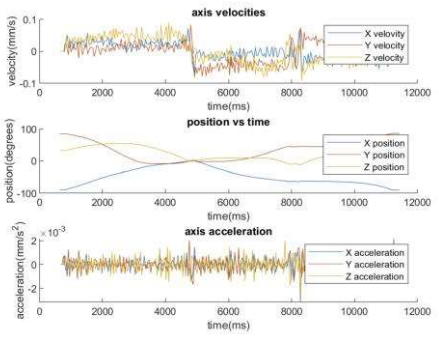


Our inverse kinematics solution is accurate, and our quintic task-space trajectory generation and trajectory following are accurate enough to complete the final project.

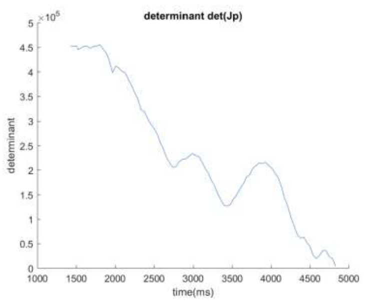
We did not use velocity kinematics in our final project, but the same trajectory as shown before is showm below, this time followed by a velocity trajectory moving in the direction of the target points:



As well as the axis velocities, X,Y,Z position, and axis acceleration for the same:



We implemented a singularity detection algorithm based on the determinant of the top half of the jacobian, which came in handy to prevent such configurations in our pick-and-place system. Below is an example graph of the determinant det(Jp) as the robot approaches a singular configuration:



These parts integrated together allowed us to create our pick-and-place system which, when calibrated properly for ambient lighting, accurately and consistently delivered all balls to their target locations regardless of how many of each color existed on the field, or where or when they were moved or removed on or from the same field.

VI. DISCUSSION

Overall, our robot exceeded our expectations for accuracy of detection and relocation. Initial issues with camera calibration and centroid localization cause us some concern, but we eventually realized that this was no fault of our own- the camera we were working with in fact had a physical flaw (possibly a deformed lens) which caused excess distortion in certain parts of the image. Once we switched to a better camera, our object localization improved dramatically.

Our code was designed from the ground up to revolve around asynchronous execution and parallel processing- while that made some tasks significantly easier (elegant dynamic object tracking was the main goal of this design), lack of experience with MATLAB's unique threading implementation caused some implementation headaches. Particularly, our camera refused to accept any communication from MATLAB's parallel pools, which was difficult to diagnose at first as the threads would simply die silently and report nothing. Once we implemented an acceptable method of debugging in this parallel environment (printing to files in try/catch blocks), we were able to determine the source of the issue and implement a workaround. In our case, this was simply to stream our camera's input through an OBS virtual camera, and have MATLAB read from OBS instead of directly from the hardware. While this technically increases latency, it is on the order of a few milliseconds which was perfectly acceptable for our use-case. Further development would likely necessitate either a switch to another type of camera, or a different program structure to work around this issue.

The portions of our program focused on kinematics worked flawlessly, with error well within our acceptable bounds for this application. We believe much of said error can be attributed to imperfections in the robot hardware and firmware, so if further development required reducing said error even further, more in-depth changes may be required.

Object localization was less precise, partly due to its nature as a complex problem, and partly due to inconsistent laboratory lighting conditions. Proper detection required recalibrating our color masks every lab session to account for changes in ambient lighting. We initially experienced issues with erroneously detecting already sorted balls as valid targets- we solved this by simply creating paper buckets for the arm to drop balls into, which obscured them from the camera. Further development in this area would primarily focus on identifying optimal and replicable lighting scenarios, and improving pre-processing to account for more widely-varying lighting conditions. This is a notable area where significant optimizations are possible- several joint angle offsets and centroid offsets which are hard-coded in our implementation would ideally be computed based on parameters detected during motion, and centroid localization could use significantly more advanced algorithms to correct for variations in the quality of color masking. It may even be found that color masking is not an optimal approach for this task, or at least that a wider variance of masks needs to be used, and the results averaged, to improve confidence of object detection and localization.

Quintic task-space trajectory planning was chosen as our primary trajectory generation due to its greater accuracy over cubic trajectories, and the fact that task-space trajectories are easier to visualize while not causing enough of a hit to performance for us to need to worry about our system performance. This decision was aided by our earlier design decisions, as offloading image processing to a separate thread allowed us to allocate more resources to trajectory generation without causing performance problems.

Velocity kinematics, while interesting, were not utilized in the final iteration of our robot. Velocity-based trajectories proved to be less precise than either cubic or quintic trajectories in the task or joint space without providing any tangible benefits. The Jacobian, however, proved useful in detecting and avoiding singular positions which would cause our inverse kinematics to fail.

Further development of our understanding of kinematic chains and robotic manipulations would be best served by experimenting with arms that have more degrees of freedom, as six degrees of freedom are required to reach all positions and orientations within the workspace. Working with a more dexterous end-effector would also enable more precise and intricate operations within the workspace.

VII. Conclusion

In this lab, we combined the skills we have been working on for the duration of the term. We combined our use of velocity kinematics to catch concerning inputs that involve singularities, inverse kinematics and trajectories to move the robot arm to positions based in joint space and task space, forward kinematics in order to find the position and orientation of the end effector, and the general system and architecture of the Hephaestus arm. Now we have fully implemented our pick and place system for the robotic arm. The arm is now able to pick up colored orbs and sort them into their respective buckets, one for each color, as well as identify objects being added to or removed from the board mid-sort. Our arm also allows for multiple of the same color orbs at the same time and will reliably sort them into their buckets. Our arm is also able to pick up random objects, which we demonstrated using a green dry-erase marker cap and placing it into the bucket for green orbs. Our robot can also dynamically follow an orb around the board as we move it. Overall, his project has been extremely useful in developing our knowledge of kinematic chains and robotic systems in general.

VIII. Link to Code

https://github.com/RBE300X-Lab/RBE3001_Matlab17/releases/tag/lab5_submission

IX. Link to Video

https://drive.google.com/file/d/1buryAWfrTO7wtXSdkQFn7508YFOWPFxr/view?usp=sharing

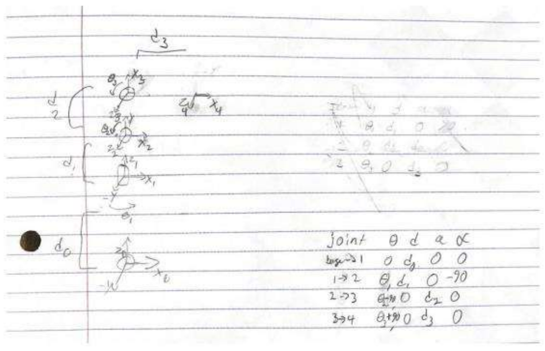
X. Table of Contributions

Name	Planning	Code	Experim.	Analysis	writeup	video
Matthew Gomes	33%	70%	40%	50%	40%	5%
Omri Green	33%	20%	25%	25%	10%	90%
Grace Holden	33%	10%	35%	25%	50%	5%

XI. Appendix

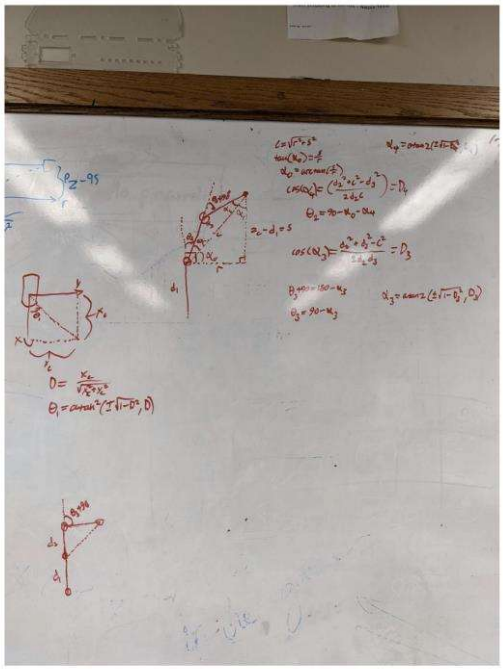
XII. Derivation of Forward and Inverse Position Kinematics

Forward kinematics:



joint	θ	d	a	α
base→1	0	d_0	0	0
1→2	θ_1	d_1	0	-90
2→3	θ_2+90	0	d_2	0
3→4	θ_3+90	0	d_3	0

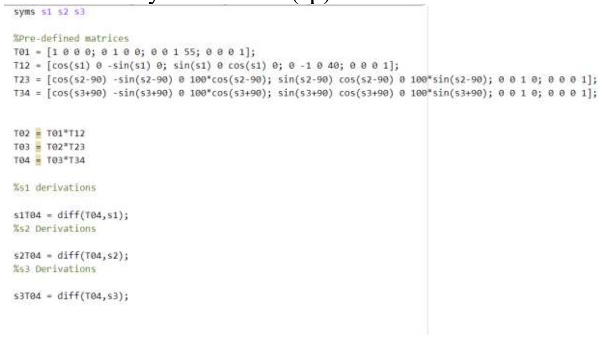
Inverse kinematics:



$P_z - 95$
$c = \sqrt{r^2 + s^2}$
$\tan(\alpha_0) = \frac{s}{r}$
$\alpha_0 = \arctan(\frac{s}{r})$
$\cos(\alpha_4) = \left(\frac{d_2^2 + c^2 - d_3^2}{2 d_2 c}\right) = D_4$
$\alpha_4 = \operatorname{atan2}(\pm\sqrt{1-D_4^2}$
$\theta_2 = 90 - \alpha_0 - \alpha_4$
$z_c - d_1 = s$
$\cos(\alpha_3) = \frac{d_2^2 + d_3^2 - c^2}{2 d_2 d_3} = D_3$
$\theta_3 + 90 = 180 - \alpha_3$
$\theta_3 = 90 - \alpha_3$
$\alpha_3 = \operatorname{atan2}(\pm\sqrt{1-D_3^2}, D_3)$
$D = \frac{x_c}{\sqrt{y_c^2 + x_c^2}}$
$\theta_1 = \operatorname{atan2}(\pm\sqrt{1-D^2}, D)$
$\theta_3 + 90$
d_1
d_2

XIII. DERIVATION OF FORWARD AND INVERSE VELOCITY KINEMATICS

Forward velocity kinematics (Jp):



```
syms s1 s2 s3

%Pre-defined matrices
T01 = [1 0 0 0; 0 1 0 0; 0 0 1 55; 0 0 0 1];
T12 = [cos(s1) 0 -sin(s1) 0; sin(s1) 0 cos(s1) 0; 0 -1 0 40; 0 0 0 1];
T23 = [cos(s2-90) -sin(s2-90) 0 100*cos(s2-90); sin(s2-90) cos(s2-90) 0 100*sin(s2-90); 0 0 1 0; 0 0 0 1];
T34 = [cos(s3+90) -sin(s3+90) 0 100*cos(s3+90); sin(s3+90) cos(s3+90) 0 100*sin(s3+90); 0 0 1 0; 0 0 0 1];


T02 = T01*T12
T03 = T02*T23
T04 = T03*T34

%s1 derivations

s1T04 = diff(T04,s1);
%s2 Derivations

s2T04 = diff(T04,s2);
%s3 Derivations

s3T04 = diff(T04,s3);
```

Inverse velocity kinematics were calculated using the pseudoinverse of the jacobian (inv(Jp)).

ACKNOWLEDGMENT

We would like to thank our professor and all of the 3001 SAs for dedicating their time to teaching us the skills and techniques required to complete this project.

REFERENCES

[1] WPI RBE 3001 course notes
[2] WPI RBE 3001 SAs
[3] SolidWorks URDF tutorial: https://www.youtube.com/watch?v=ge3P307TgJI
[4] Misc. MATLAB documentation and user forum posts