**Project 1:** DH Parameters and Optimizations



**Author:** Siva Chinthalapudi

**Course:** BME 5425

**Description:** Introduction to Robotic Systems I

**Partner:** Rocky Saxena

### Table of Contents

Table of Contents

[Abstract 3](#_bookmark0)

[Introduction 3](#_bookmark1)

[Mathematical Concepts of Robotics 3](#_bookmark2)

[Rotations and Transformations 3](#_bookmark3)

[Forward Kinematics and Inverse Kinematics 6](#_bookmark4)

[Denavit-Hartenberg Notation 7](#_bookmark5)

[Methods 8](#_bookmark6)

[Hardware and Measurements 8](#_bookmark7)

[Coding 11](#_bookmark8)

[Results 15](#_bookmark9)

[Discussion/Conclusion 16](#_bookmark10)

[Appendix 16](#_bookmark11)

[Graphic user interface code 16](#_bookmark12)

[Objective function code 23](#_bookmark13)

[References 24](#_bookmark14)

# Abstract

The development of robotics has spread fear to many individuals in the working force, panic of losing one’s job causes a great deal of stress and feelings of hopelessness sets in. This paper explains that development of robotics will get rid of dangerous labor intrusive jobs and will allow for individuals to retrain themselves in this emerging field. The understanding of mathematical concepts of robotics that include inverse kinematics, forward kinematics, transformation matrices, rotational matrices, and dh parameters will allow anyone to build up a mathematical model of their very own robotic system. Additionally, basic hardware is utilized to generate our robot and an idea of object-oriented programming was used simulate our robot. Development of the graphic user interface (GUI) allowed for us to move our robot via slider controls and display the end effector position of the robot in real-time. Additionally, the use of forward kinematics allowed us to optimize our dh parameters to give a more precise model of our robot. These basic concepts allow anyone interested in the field of robotics to retrain themselves to solve complex re-world problems. A YouTube link is provided to show the result of our robot <https://www.youtube.com/watch?v=JZ6UTloXnIE>

# Introduction

There has been a strong push for robotics in the modern world especially due to the advancements in both hardware and software. The rise of robotics has spread fear to many individual who believe that their jobs are at risk of being taken over by these robotic systems, yet many fail to realize that development of robotics systems will take away the hard menial labor jobs that promote risk of injury to workers, this will allow society to collectively solve harder problems that require a lot of quantitative, creative, and analytical problem solving skills that many robotic systems will not be able to do. It could be argued that an Artificially Intelligent system can also do these complex problems that include researching the protein folding problem or solving an advance math formula, however, in our current state artificial intelligence is still in its infancy and the prospect of producing a generalized system will take many years to achieve.

This paper will allow people who are interested in leaving their menial jobs and seek knowledge of building robotic systems at the very basic level (complexity of robotic design can get difficult, but this paper will introduce only the basic concepts of robotics). The only pre-requisite knowledge required is some general knowledge of calculus, linear algebra, and basics of programming. Additionally, this paper will explain the theoretical knowledge of building robots and an implementation of it using MATLAB software tools by showing steps in our groups process in building our first robot. The aim of the paper is to allow the reader to learn the basics of robotics and build their own robotic model to get started in a career in robotics.

# Mathematical Concepts of Robotics

## Rotations and Transformations

The math that is involved in building your first robot includes transformation matrices. A robotic system is built with links and joints and at each joint there is a 3-dimensional coordinate system. A description of this is given in **figure 1**. In order to allow a robotic arm to move, each position and orientation of a coordinate system must be known relative to one another. The process of knowing the position of one coordinate system say frame A relative to another coordinate system (frame B)is known

as a **transformation**. Before, diving into the transformation aspect we must look at how coordinate systems are related to one another using rotational matrix. The example given is taken from Dr.

Pandya’s lecture slide. If we are given two coordinate systems named frame A and frame B and we want to know what frame A is relative to frame B we first set up an identity matrix.

x y z [1 0 0

0 1 0

0 0 1]

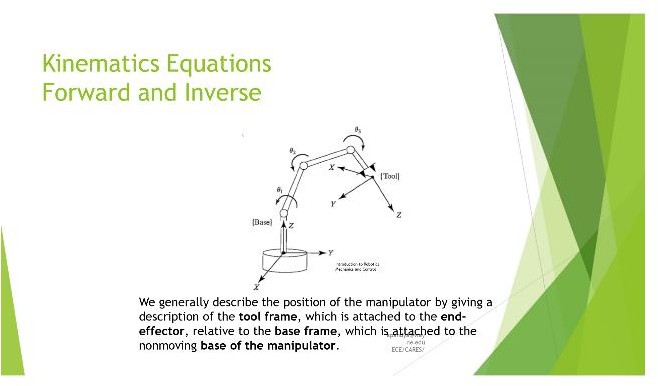
Where x, y, and z represent the position of the axes in the column view. The question is if we rotate frame A relative to frame B what is going to be frame A in frame B. In our example we do a 90-degree rotation about the z axis (note when we rotate around the z axis the z axis will REMAIN the same, similar if we rotated about x, x would stay the same). Now by applying the right-hand rule thumb points to z, index finger to x, and middle to y, we look down the z axis and we rotate. An, extremely important concept is that if you rotate +z you rotate counter-clockwise and -z is clockwise. Now with this rotation we look at the rotation and compare it with the old frame A and see where our new values line up in relation to before the rotation, which produces a new matrix.

x y z [0 1 0

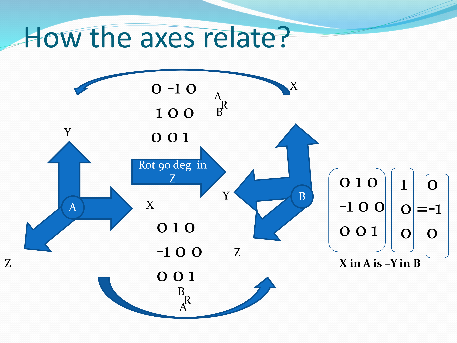
-1 0 0

0 0 1]

**Figure 2** gives and excellent visual description of the process.



**Figure 1:** visual description of the robotic coordinate system of a manipulator.



**Figure 2:** Visual description of rotation about an axe in one frame to another

Next, the question that needs to be addressed is what if we have a point in frame B and we wish to know this same point in a new frame say frame A. Since we know what rotations are, all we have to do is take it one step forward to get our point in another frame. The example given was the same example that Dr. Pandya lectured on. Say that frame B has been rotated relative to frame A about Z axis by 30 degrees (remember rotation about z axis will still make z axis in the same position). The rotated matrix produced is as followed

[0.866 -0.500 0.00

0.500 0.866 0.00

0.000 0.000 1.000]

Now we wish to figure out the **translation**, the translation essentially means how far in the x, y, and z direction our frame B is to frame A. If we wish to translate 10 units in the x direction and 5 units in the y direction, adding this to our above already rotated matrix of frame B to frame A gives

[0.866 -0.500 0.00 **10.0**

BtoA = 0.500 0.866 0.00 **5.0**

0.000 0.000 1.000 **0.0**

0.000 0.000 0.000 **1**]

By observing the bold numbers, you see all we did was add it to our rotation matrix. Note the extra row of 0 0 0 1 was added to allow for matrix multiplication, the new row adds 0 weight in our calculations and can be ignored. The given point in frame B is

[3.0

BP = 7.0

0.0]

The final step is simply doing matrix multiplication to yield point B in A frame by

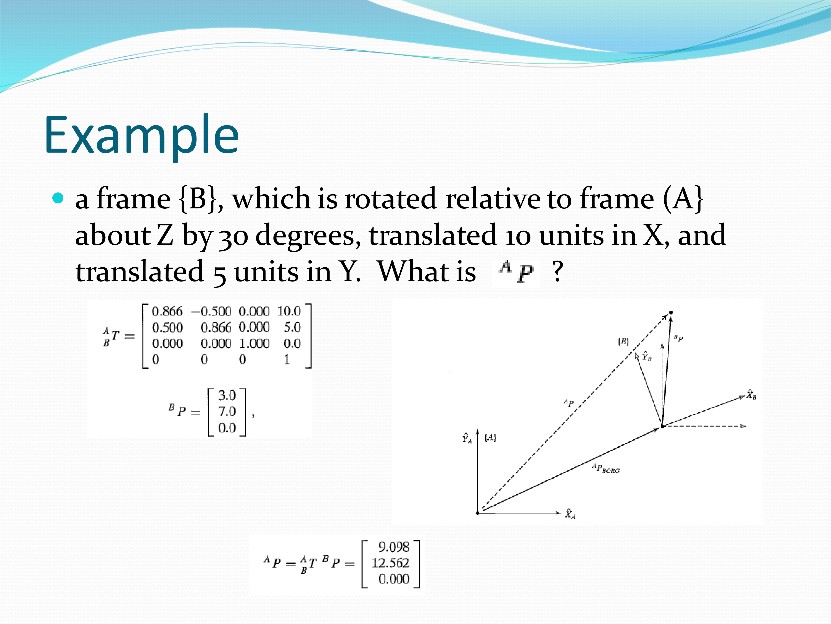
AP = BtoA \* BP

which will give us the new point in frame A as [9.098

12.562

0.000]

Note since this is matrix multiplication order matters so make sure the correct orientation is used. The use of matrix multiplication is an excellent visualization that shows that sets of equation are simultaneous solved together. **Figure 3** gives an excellent visualization of the process of transformation

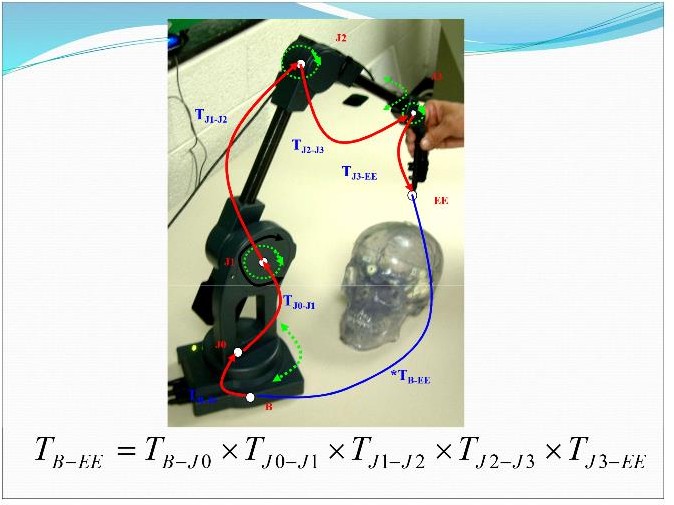


**Figure 3:** Transformation visualization

## Forward Kinematics and Inverse Kinematics

The question that needs to be addressed is what is the purpose of these transformation matrices? We understand how to calculate them but what can we do with them? The interesting answer is that if we know the transformation from each coordinate system, we can find the position and orientation relative to the other. For example, if we know the position of our base how do we know the position of our end effector? The answer is simply by transformation preferably **forward kinematics**. All we would have to do is simply keep transforming from each frame to each frame and it will generate our transformation matrix of the end effector. So, by knowing the joint angles we get the transformation at the end effector. **Figure 4** gives an excellent visual representation of this process. By now the thought will arise in which case what if we only know the position of the end effector how will we know the position of the end effector relative to the base? This can also be achieved by **inverse kinematics**. In inverse kinematics we can find the position of the base frame simply by inputting our transformation of the end effector that will give us all of our joint variables needed to figure out the position of the base coordinate system. Now we understand rotations, transformations, forward kinematics, and inverse

kinematics, how do we generate these transformation matrices? This will be explained in the next section on Denavit-Hartenberg (DH) Notations.



**Figure 4:** Practical use of applying transformations to yield position of end effector

## Denavit-Hartenberg Notation

The Denavit-Hartenberg Notation (dh parameters) are used to get our specified parameters for our joints and links, by obtaining these specified parameters we are able to generate the transformation matrices and explain the architecture of our kinematics chain (the links and joints). The joint parameters and variables are as followed.

Link Length (ai): This is the distance between the old z axis and the new z axis.

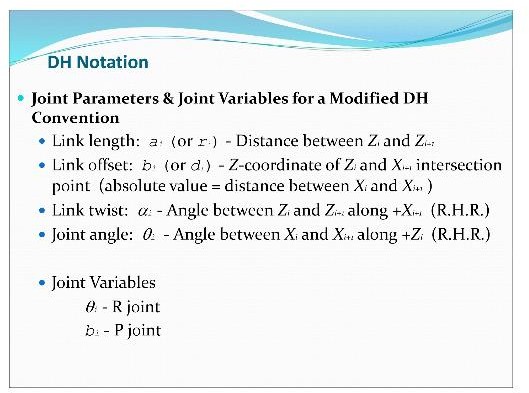
Link offset (bi): This is the z coordinate of the old z axis and the new x axis at the point they intersect.

Link Twist (alphai): This is the angle between the old z axis and the new z axis when you look down the new x axis.

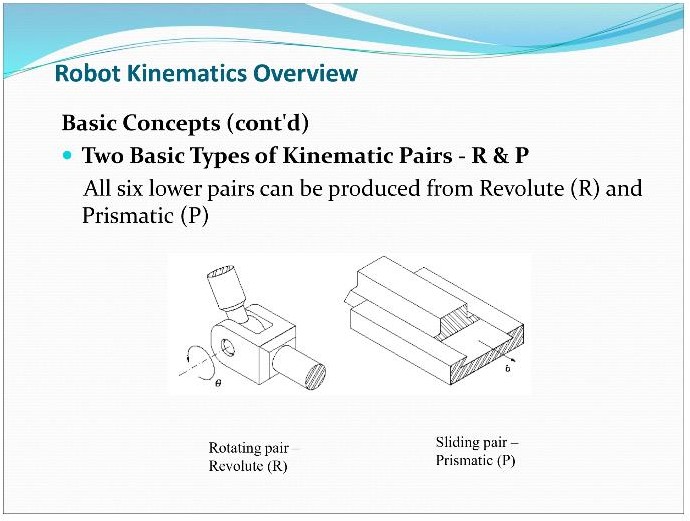
Joint angle (theta): The angle between the old x axis and the new x axis while looking along the old z axis.

Please refer to **figure 5** for the table explaining how to find the dh parameters.

In any condition there will always be 3 parameters (values that do not change) and one variable type (a changing condition). The variables will either be the joint variable theta if the joints rotate and link offset if the robot exhibits a sliding movement. **Figure 6** gives a visualization on how this movement occurs.



**Figure 5:** How to find the dh parameters.



**Figure 6:** Visualization that determines which variable your dh parameter has.

**Figure 6** explains which variable is in your dh parameter based on the kinematics chains of your robot. Note that if your robot exhibits a rotation or revolute than the variable will be theta and the link offset will be 0, if your robot exhibits sliding or prismatic motion than the variable in your dh parameter is the link offset and theta is 0. Obtaining these dh parameters is a powerful tool because once obtained running the robotics tool kit explained later in the section will allow us to generate a robotics model.

Note one thing to keep in mind is that for each joint in your robot you must measure the dh parameters for each joint.

# Methods

In this section we will apply the mathematical concepts discussed in previous sections to help us build our own 3 link robotic arm. This is where the exciting aspect comes because it allows us to bring all the mathematics involved to generate a moveable robotic arm. Additionally, this section will be split into the hardware/measurement set up and software that was used in the process.

## Hardware and Measurements

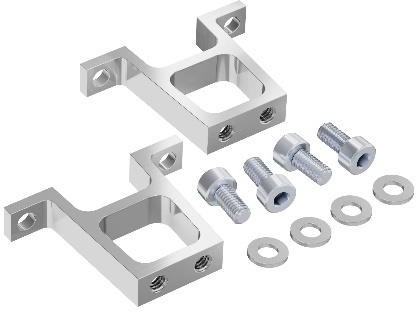
The following hardware materials were used

2 x HS-311 Standard Servo Motor 2 x Aluminum Beams 4.62” 1 x HS-81 Micro Servo Motor

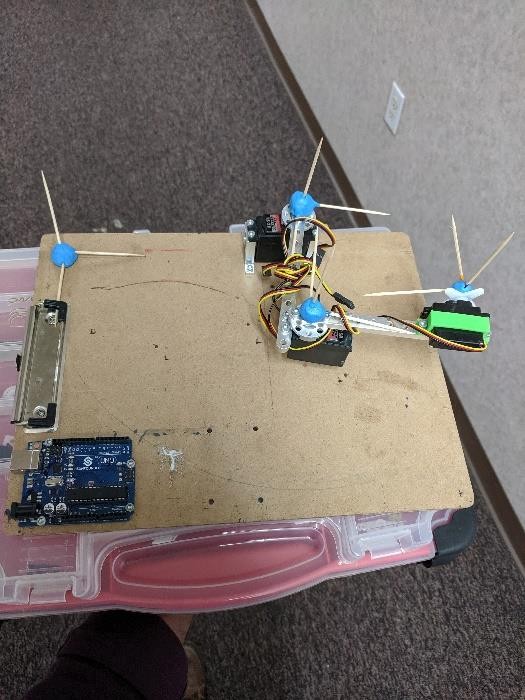


9 toothpicks Blue sticky tack 1 x clipboard



2 vertical mounts with 4 screws and 4 washers

Additionally, one fan was used in the process. **Figure 7** provides a visualization of our robotic system.



**Figure 7:** Visualization of robotic arm

The steps involved in setting up the hardware of the robotic arm are as followed.

1. Creating holes with a drill to maximize the range of motion of the robotic arm. This was accomplished simply by testing all different range of motions the robotic arm can move in different locations on the clipboard. Once finding optimal range of motion 4 holes were drilled.
2. Once the holes were drilled a single motor was screwed in via brackets.
3. Next a link was added to our mounted rotor, we reduced a single hole on the link by one because it was the last digit of my access id (eh2971).
4. The link was attached from the previous rotor to the new rotor that is suspended in air via screws.
5. HS-81 rotor was attached to the end of the link via a green bracket. Additionally, a fan was attached to the HS-81 rotor to simulate the end effector
6. 3-dimensional coordinate systems were made by grabbing 3 toothpicks (x, y, and z) and morphed together using blue sticky tack. These coordinate systems were attached to each joint.
7. A universal coordinate system was also developed on the side of the clip board to reference our robots base and end effector tool tip to it.

Now that the robot is mounted the next step was to provide our measurements. Measurements were taken for the joint limits by using a protractor, extending the range of each joint from the negative and positive end gave the maximum and minimum limits of each joint, this will be our qlims for the robotics tool kit. The base of the robot was measured via a ruler from the universal coordinate system, the distance was 8 inches in the y direction and similarly the tool tip of the robot was measured in the downward z direction. Next the dh parameters were measured via using the method in **figure 5** using a ruler and a protractor, refer to **figure 8**. The link offsets were measured from z axis to z axis as visualized in the **figure 8** below. The final measurements were used for our optimization process, we measured actual values from the base to the end effector by setting 5

pairs of our 3 joint angles. The full optimization process will be discussed in the coding aspect. Since all the hardware is mounted and all measurements are obtained, the coding aspect will directly model our robot into MATLAB.

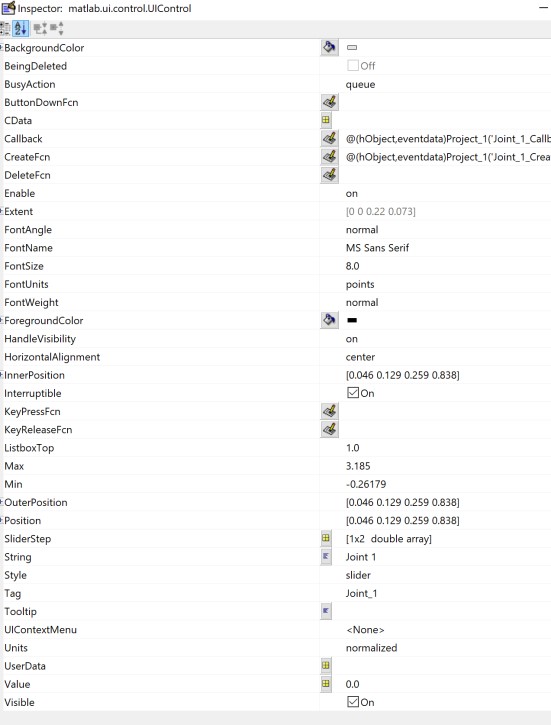


## Coding

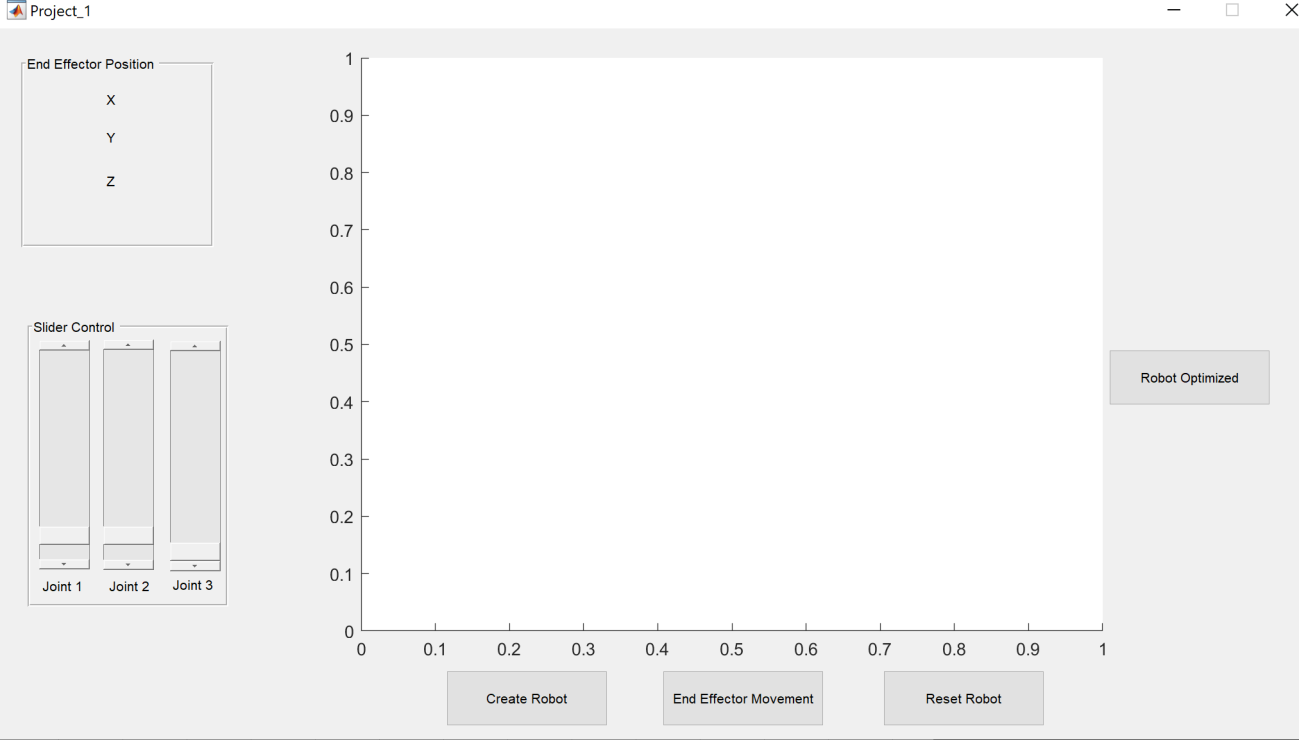
**Figure 8:** visualization how dh parameters were obtained.

The coding was done by using Peter Corke’s robotics tool kit, follow the link on here on how to download the tool kit <https://petercorke.com/wordpress/toolboxes/robotics-toolbox>. The first step we must do is to develop a graphical user interface (GUI), this is done by typing in guide in the command window of your project. Next, we create 3 sliders for each joint variable (theta 1, theta 2, and theta 3). Note this will create 3 call back functions and we must change the tag of each joint variable, so we know which function refers to which slider. The tags were changed to joint\_1, joint\_2, and end\_effector. The axes were created to provide an environment for our robot. Several other buttons were also created including create robot (creates the physical robot), end effector movement (moves the end effector), reset robot (places the robot back to initial starting position), and robot optimized which will use an fmin search function to reduce the errors to allow us to get better dh parameters for our robot.

Additionally, an end effector position static text will be displayed in the corner to update the position of our end effector as we move the joint slider. Refer to **Figure 9** for updating tags and **figure 10** for the layout of the graphic user interface.



**Figure 9:** Property inspector to change tags for function call backs. Right click on the slider to open the inspector. Additionally, strings can be changed to pick the name of your sliders or button.



**Figure 10:** Layout of GUI.

Once the layout of the gui is created our next step is to update the coding involved in each function to allow our robot to move. The opening function is the function that has variables that will be accessible to all other functions. We first initialize our robot with handles.robot = []. As of right now we

have an empty robot. The handles aspect is considered a struct which is part of the object-oriented programming (OOP) concepts, what this means is that handles will have a set of variables and methods that it can call. So when we add handles.theta = [0,0,0] this will give us a vector of theta 1, theta 2, and theta 3 values. At this point the variables of the handles struct are robot and theta. The slider function call back named Joint\_1 (note how we changed the tag to this) is how we will change our theta 1 for our joint one value. handles.theta(1) = get(hObject,'Value'); This portion of the code gets the value of the slider as we move the slider and store it into our theta(1) position of our vector. handles.robot.plot (handles.theta); will plot our robot with the updated theta 1 value. The following code

T = handles.robot.fkine(handles.theta); X = sprintf('X = %f', T.t(1)); set(handles.x\_func,'String',X); drawnow();

is going to be used to update our x position of our end effector. Note how in the section in the mathematics concepts we explained what forward kinematics does, this is why learning the mathematics allows us to build great robots. What fkine does is that we will insert our joint angles (theta vector that has theta 1, theta 2, and theta 3) and it will display a transformation matrix of the end effector. From there the matrix will provide the x, y, and z position of the end effector, and using set handles.x\_func (static text tag) we will store this image to be displayed in our static text box. These same steps will be applied in the Joint\_2 and End\_Effector call back functions as well. The reader should note that it is of utmost importance to put the following code

guidata(hObject, handles);

after each function, failure to do so will cause our theta values to be 0. The reason why they will be 0 is because in the opening function we initialized the theta values to be 0, if we did not provide the code above then our theta values would only be localized to each function and not be accessible to any other function. This is a computer science concept known as function stacks, each function has its own stack that has localized variables that are only accessible to them and not any other functions, in other programming languages you can pass variables by reference or values and if we pass by value it means the variables values are only retained in that specific function. Passing by reference will allow the variable to be updated in each function it is sent to and in essence the above code line does this.

The large portion of our code is dedicated to the Create\_Robot call back function, this is the function that will use our dh parameters we measured to build our robot. The following code

L(1) = Link('a', 3.65,'alpha', 0, 'd', 1.75, 'offset', pi/18, 'qlim', [-160 160]\*deg);

will generate our link by simply taking our dh parameters. This function will be applied to generate link 2 and 3 as well. Note when we have some variable and it has a () aspect it is generating a vector of values. So, the variable L will have 3 values for link 1, 2, and 3. Each of these values can be accessed simply by using L(1), L(2), and L(3) to return the values at each index position. The SerialLink function will take the following parameters, the link we generated above with our dh parameters and name of our robot

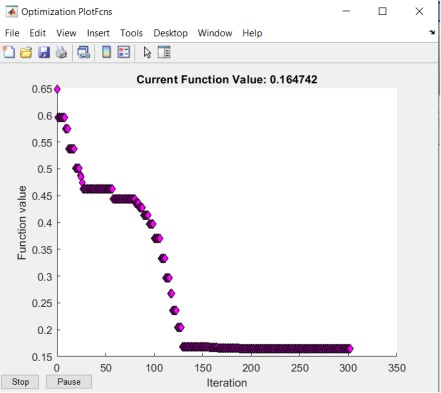
which is a string value hence the ‘ ‘ symbols. The serial link will essentially represent the links and joint variables according to the robotics tool kit manual.

The following code will be used to generate the base and tool tip position of our robot which was physically measured from the universal coordinate system via a ruler in inches. This was described in the hardware and measurement section.

handles.robot.base handles.robot.tool

Once we obtained our robotics base position, tool tip position, links, and have our theta values obtained by the Joint call back function (as we move the sliders) we can plot our robot. The function that allows us to change our end effector is named the Key\_press\_KeyPressFcn. The best way to understand this code is looking up what the ascii table is. Essentially the ascii keys are used because the computer only understands number, so generally if you press an arrow key the computer cannot recognize it, therefore, developing numbers for each key pressed on the keyboards was established in the ascii table. Once an arrow key is pressed it is assigned a specified number from this ascii table and will be converted into binary for the computer to understand (although you don’t have to know the binary representation), so the number will be passed into a switch statement and carry out the designated case to move the x, y, and z position of your robot. A question may arise as to how we got this position for x, y, and z in which case it was solved by using forward kinematics that obtained the transformation matrix with the specified coordinates of the end effector. This function is also where we will utilize ikine function which will get the new joint angles based on the movement of the end effector storing them into our handles.theta struct variable.

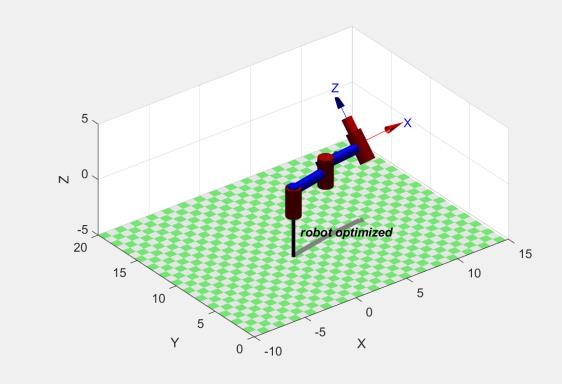
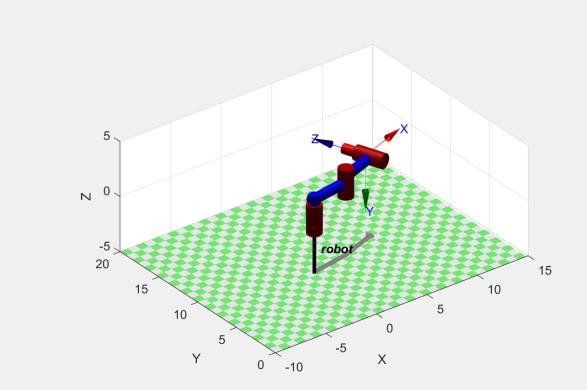
The final function that was written in the gui was the Optimize\_fn which utilized a separate script file written in MATLAB called objFunc. What happens is we initialize our dh parameters obtained from **figure 7** and pass them in as an argument to the Optimize\_fn, once passed in the code enters the Optimize\_fn and begins running that function. Inside this external script we initialize 5 sets of joint angles for our theta values into a variable called data, the purpose of data is to generate the actual measurements of our distance of the base to the end effector given specified joint angles. Next, we run a for loop that will iterate 5 times and we run the fkine function that will be used to get our transformation matrix based off the joint angles set in our data variable, this will give us our transformation matrix at the end effector in which case will give the x and y position of end effector (the z position is not relevant in this instance). To obtain the distance we use the Pythagorean theorem x^2 + y^2 = d^2 to get out predicted distance. The last step is simply applying the Euclidean distance to observe the predicted error obtained from the transformation to the actual distance measured physically with a ruler. Once the objFunc ends we enter back into our Optimize\_fn and we apply the fminsearch function which will find the minimal error from our dh parameter guess and return the new 5 sets of joint angles where the minimal amount of error is observed. Once the optimized dh parameters are obtained, we plot the new optimized robot based off the steps in the create function. **Figure 11** shows a graph of fminsearch and objFunc decreasing our error.



**Figure 11:** visualization of the error reaches a minimum threshold set by fminsearch

# Results

Once the hardware and coding aspect of the project are done it is time to evaluate our robotic arm. The gui has buttons to create our robot and using sliders which will allow us to move each theta angle of our robot (revolute variable). The reset button was provided to reset our robot to initial conditions which is when our theta values are all 0. There also is a static text that will update our robotic arm in real time as we move each joint slider around which is all due to forward kinematics that gives the end effector position. Additional functionality was implemented to move the end effector position by using the keyboard arrows due to the use of ascii table. The final piece that ties everything together was the robot optimized function that will plot an optimized version of our robot based of the new dh parameters obtained by fminsearch and objFunc. This optimization will reduce the error caused by fminsearch to generate as close to actual measured distance between the end effector and base coordinate system. **Figure 12 and figure 13** show a before and after visualization of our robot when dh parameters where not optimized compared to when they were optimized.



**Figure 12:** unoptimzed robot. **Figure 13:** Optimized robot

# Discussion/Conclusion

It is true that the development of robotics will certainly take away jobs from people, however, fear should not be experienced from these individuals. This project report showed that with a little bit of mathematics, some coding skills, and an aptitude to learn many individuals can retrain themselves to solve complex problems instead of simply bagging groceries. The use of forward kinematics, inverse kinematics, dh parameter notations, transformation matrices, and numerical methods (optimization) allows anyone to build simulation of a robotic arm. This basic tutorial shows that by understanding the mathematical concepts of robotics allows anyone who feels threaten that they will not be relevant in the work force to feel inspired to be part of the many individuals solving complex real-world problems.

The basic idea is if we can get rid of these manual labor/dangerous jobs then we as humans can focus our time on solving complex problems that we may have not had the chance to explore since we were busy doing these less mentally challenging menial labor jobs. The complexity of problems can be associated in biomedical engineering disciplines that include creating robotic surgery to allow surgeons to have precision techniques to avoid nerve damages that can potentially paralyze patients. The use of robotics can be utilized in space exploration by sending robots to planets that will sample geological data (mars rover), places were humans unfortunately cannot reach due to distance and biological constraints. The final mention where robots can be utilized is in an area that is critically growing which is autonomous vehicle systems. The ability to develop self-driving cars will prevent car crashes and allow elderly people to go from places to places without having to rely on family members or a health care support system.

The future is certainly bright for the development of robotics and in the end, people should be excited of this emerging field not frighten. The development of robotics will allow for new challenging problems to be solved and the ability to understand a little mathematics, coding, and hardware will allow anyone to build their very own robot. The future looks promising and steps should be taken in training oneself to stay relevant in todays workforce and solve problems that can push the boundaries of all human knowledge.

# Appendix

## Graphic user interface code

function varargout = Project\_1(varargin)

% PROJECT\_1 MATLAB code for Project\_1.fig

% PROJECT\_1, by itself, creates a new PROJECT\_1 or raises the existing

% singleton\*.

%

% H = PROJECT\_1 returns the handle to a new PROJECT\_1 or the handle to

% the existing singleton\*.

%

% PROJECT\_1('CALLBACK',hObject,eventData,handles,...) calls the local

% function named CALLBACK in PROJECT\_1.M with the given input arguments.

%

% PROJECT\_1('Property','Value',...) creates a new PROJECT\_1 or raises the

% existing singleton\*. Starting from the left, property value pairs are

% applied to the GUI before Project\_1\_OpeningFcn gets called. An

% unrecognized property name or invalid value makes property application

% stop. All inputs are passed to Project\_1\_OpeningFcn via varargin.

%

% \*See GUI Options on GUIDE's Tools menu. Choose "GUI allows only one

% instance to run (singleton)".

%

% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help Project\_1

% Last Modified by GUIDE v2.5 12-Oct-2019 18:02:53

% Begin initialization code - DO NOT EDIT gui\_Singleton = 1;

gui\_State = struct('gui\_Name', mfilename, ... 'gui\_Singleton', gui\_Singleton, ... 'gui\_OpeningFcn', @Project\_1\_OpeningFcn, ... 'gui\_OutputFcn', @Project\_1\_OutputFcn, ... 'gui\_LayoutFcn', [] , ...

'gui\_Callback', []); if nargin && ischar(varargin{1})

gui\_State.gui\_Callback = str2func(varargin{1}); end

if nargout

[varargout{1:nargout}] = gui\_mainfcn(gui\_State, varargin{:}); else

gui\_mainfcn(gui\_State, varargin{:}); end

% End initialization code - DO NOT EDIT

% --- Executes just before Project\_1 is made visible.

function Project\_1\_OpeningFcn(hObject, eventdata, handles, varargin)

% This function has no output args, see OutputFcn.

% hObject handle to figure

% eventdata reserved - to be defined in a future version of MATLAB

% handles structure with handles and user data (see GUIDATA)

% varargin command line arguments to Project\_1 (see VARARGIN)

% Choose default command line output for Project\_1 handles.output = hObject;

% the robot initalized to 0 handles.robot = [];

% Robots joint variables handles.theta = [0, 0, 0];

% This code is needed to make sure our variables retain there values

% in other functions guidata(hObject, handles);

% UIWAIT makes Project\_1 wait for user response (see UIRESUME)

% uiwait(handles.figure1);

% --- Outputs from this function are returned to the command line. function varargout = Project\_1\_OutputFcn(hObject, eventdata, handles)

% varargout cell array for returning output args (see VARARGOUT);

% hObject handle to figure

% eventdata reserved - to be defined in a future version of MATLAB

% handles structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure varargout{1} = handles.output;

% --- Executes on slider movement.

function Joint\_1\_Callback(hObject, eventdata, handles)

% This code will grab the value of our slider position and store it to our

% structs variable theta(1) handles.theta(1) = get(hObject,'Value');

% with the new theta value we will plot our robot with the new angle handles.robot.plot (handles.theta);

% The code below will be used to update end effector x position in

% real time as we move the slider

T = handles.robot.fkine(handles.theta); X = sprintf('X = %f', T.t(1)); set(handles.x\_func,'String',X); drawnow();

% Update handles structure guidata(hObject, handles);

% --- Executes during object creation, after setting all properties. function Joint\_1\_CreateFcn(hObject, eventdata, handles)

% hObject handle to Joint\_1 (see GCBO)

% eventdata reserved - to be defined in a future version of MATLAB

% handles empty - handles not created until after all CreateFcns called

% Hint: slider controls usually have a light gray background.

if isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor')) set(hObject,'BackgroundColor',[.9 .9 .9]);

end

% --- Executes on slider movement.

function Joint\_2\_Callback(hObject, eventdata, handles)

% This code will grab the value of our slider position and store it to our

% structs variable theta(2) handles.theta(2) = get(hObject,'Value');

% with the new theta value we will plot our robot with the new angle handles.robot.plot(handles.theta);

% The code below will be used to update end effector x position in

% real time as we move the slider

T = handles.robot.fkine(handles.theta); Y = sprintf('Y = %f', T.t(2)); set(handles.y\_func,'String',Y); drawnow();

% Update handles structure guidata(hObject, handles);

% --- Executes during object creation, after setting all properties. function Joint\_2\_CreateFcn(hObject, eventdata, handles)

% hObject handle to Joint\_2 (see GCBO)

% eventdata reserved - to be defined in a future version of MATLAB

% handles empty - handles not created until after all CreateFcns called

% Hint: slider controls usually have a light gray background.

if isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor')) set(hObject,'BackgroundColor',[.9 .9 .9]);

end

% --- Executes on slider movement.

function End\_Effector\_Callback(hObject, eventdata, handles)

% This code will grab the value of our slider position and store it to our

% structs variable theta(3) handles.theta(3) = get(hObject,'Value');

%with the new theta value we will plot our robot with the new angle handles.robot.plot(handles.theta);

% The code below will be used to update end effector x position in

% real time as we move the slider

T = handles.robot.fkine(handles.theta); Z = sprintf('Z = %f', T.t(3)); set(handles.z\_func,'String',Z); drawnow();

% Update handles structure guidata(hObject, handles);

% --- Executes during object creation, after setting all properties. function End\_Effector\_CreateFcn(hObject, eventdata, handles)

% hObject handle to End\_Effector (see GCBO)

% eventdata reserved - to be defined in a future version of MATLAB

% handles empty - handles not created until after all CreateFcns called

% Hint: slider controls usually have a light gray background.

if isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor')) set(hObject,'BackgroundColor',[.9 .9 .9]);

end

% --- Executes on button press in Create\_Robot.

function Create\_Robot\_Callback(hObject, eventdata, handles)

% value of our work space to change the floor titles W = [-10 15 0 20 -5 5];

% deg will be used to convert degrees to radians deg = pi/180;

% By inputting our dh parameters as an argument we will generate each link L(1) = Link('a', 3.65,'alpha', 0, 'd', 1.75, 'offset', pi/18, 'qlim', [15 -182.5]\*deg);

L(2) = Link('a', 3.6, 'alpha', -(pi/2) , 'd', 0.525, 'offset', pi/18, 'qlim', [182.5 -15]\*deg);

L(3) = Link('a', 0 , 'alpha', 0, 'd', 0.45, 'offset', 0, 'qlim', [0, pi]);

% serial link will be used to help create the robot handles.robot = SerialLink(L, 'name', 'robot');

% robotic base that is 8 inches in y direction from universial coordinate

% to robotic base, similarly for the tool tip handles.robot.base = [1 0 0 0; 0 1 0 8; 0 0 1 0; 0 0 0 1];

handles.robot.tool = [1 0 0 0; 0 1 0 0;0 0 1 -2.5; 0 0 0 1];

% below code will plot our robot with the theta values and update

% the floor of our robot with the 'workspace' command handles.robot.plot (handles.theta, 'workspace', W);

% Update handles structure guidata(hObject, handles);

% --- Executes on button press in Reset\_fn.

function Reset\_fn\_Callback(hObject, eventdata, handles)

% reset the robot back to starting postion by making theta values 0 handles.robot.plot ([0 0 0]);

% --- Executes on button press in Key\_press.

function Key\_press\_Callback(hObject, eventdata, handles)

% ignore this function

% --- Executes on key press with focus on Key\_press and none of its controls. function Key\_press\_KeyPressFcn(hObject, eventdata, handles)

% hObject handle to Key\_press (see GCBO)

% eventdata structure with the following fields (see MATLAB.UI.CONTROL.UICONTROL)

% Key: name of the key that was pressed, in lower case

% Character: character interpretation of the key(s) that was pressed

% Modifier: name(s) of the modifier key(s) (i.e., control, shift) pressed

% handles structure with handles and user data (see GUIDATA)

% this code will get the current graphic figure f= gcf;

% m matrix specfies the axis (x,y,and z) and if there are any yaw, pitch, and row m = [1 1 0 0 0 0];

% val will get the current arrow key from the keyboard, note computer can

% only recongnize numbers so the ascii table assigns a number to each

% keyboard key and this number is what is stored in the val variable val=double(get(f,'CurrentCharacter'));

%do a forward kinematics

T = handles.robot.fkine([handles.theta(1), handles.theta(2), handles.theta(3)]);

% depending on the val value it will run the switch case based on it. switch (val)

case 28 % right T.t(1) = T.t(1) + .1;

case 29 %left

T.t(1) = T.t(1) - .1;

case 30 %up

T.t(2) = T.t(2) + .1;

case 31 %down T.t(2) = T.t(2) - .1;

otherwise

disp ('unrecognized key');

end

% inverse kinematics is used here to get the joint angles which are theta 1

% theta 2 and theta 3.

q = handles.robot.ikine(T,'q0', handles.theta,'mask', m);

% if q is not empty meaning there is some value in it then we will plot the

% robot and store the new joint angles in our theta values. This is good to

% prevent self assignment and decrease the time complexity of our program. if ~isempty(q)

handles.robot.plot([q(1) q(2) q(3)]); handles.theta(1) = q(1); handles.theta(2) = q(2);

handles.theta(3) = q(3); end

% Update handles structure guidata(hObject, handles);

% --- Executes on button press in Optimize\_fn.

function Optimize\_fn\_Callback(hObject, eventdata, handles)

% we first initalize our dh parameters.

dh\_parameters = [3.65 0 1.75 pi/18 3.6 -(pi/2) 0.525 pi/18 0 0 0.45 0];

% This functions calls our script file function and passes our intalized

% dh\_parameters as an argument to the function objFunc(dh\_parameters)

% The code below will display and error values for each iteration completed options = optimset('PlotFcns',@optimplotfval);

% This code will give us our optimized dh parameter based of the minimal

% error threshold obsered by fminsearch

new\_dh = fminsearch(@(dh\_parameters) objFunc(dh\_parameters), dh\_parameters, options)

% The code below is similar to the create robot function, but this time we

% are plotting our robot based on the new optimized set of dh parameters.

% value of our work space to change the floor titles W = [-10 15 0 20 -5 5];

% Pick the correct axe to plot our robot axes(handles.axes1)

deg = pi/180;

L(1) = Link('a', new\_dh(1),'alpha', new\_dh(2), 'd', new\_dh(3), 'offset', new\_dh(4), 'qlim', [15 - 182.5]\*deg);

L(2) = Link('a', new\_dh(5), 'alpha', new\_dh(6) , 'd', new\_dh(7), 'offset', new\_dh(8), 'qlim', [182.5 - 15]\*deg);

L(3) = Link('a', new\_dh(9) , 'alpha', new\_dh(10), 'd', new\_dh(11), 'offset', new\_dh(12), 'qlim', [0, pi]); handles.robot = SerialLink(L, 'name', 'robot optimized');

handles.robot.base = [1 0 0 0; 0 1 0 8; 0 0 1 0; 0 0 0 1];

handles.robot.tool = [1 0 0 0; 0 1 0 0;0 0 1 -2.5; 0 0 0 1]; handles.robot.plot (handles.theta, 'workspace', W);

% Update handles structure guidata(hObject, handles);

## Objective function code

function error = objFunc(dh\_parameters)

% The purpose of this is to get the error between our measured (actual

% values) and our predicted values generated from fkine by using the

% euclidean distance.

% data is going to be 5 sets of our joint angle positions that will be used

% to optimize the dh parameters data = [pi/2 0 0 7.25;

pi/2 pi/2 0 5.25;

0 -pi/2 0 5.25;

0 pi/6 0 7.5;

pi/6 pi/6 0 7.5];

% Based off the new dh parameters we pass them as an argument to our link

% parameter, take note of the () position corresponds to matrix above

L(1) = Link('a',dh\_parameters(1) ,'alpha', dh\_parameters(2), 'd', dh\_parameters(3), 'offset', dh\_parameters(4));

L(2) = Link('a', dh\_parameters(5), 'alpha', dh\_parameters(6) , 'd', dh\_parameters(7), 'offset', dh\_parameters(8));

L(3) = Link('a', dh\_parameters(9) , 'alpha', dh\_parameters(10), 'd', dh\_parameters(11), 'offset', dh\_parameters(12));

handles.robot.base = [1 0 0 0; 0 1 0 8; 0 0 1 0; 0 0 0 1]; handles.robot = SerialLink(L, 'name', 'robot');

%initalize error to be 0 it is an accumulator variable.

error = 0;

for i = 1:length(data)

% fkine will give our predicted transformation matrix that holds our

%x and y position of the end effector

predict(i) = handles.robot.fkine([data(i,1), data(i,2), data(i,3)]);

% using the pythagorem theorm to get the distance based off x and y predict\_pythagreom(i) = sqrt((predict(1,i).t(1))^2 + (predict(1,i).t(2))^2);

% this code will sum all the error with each iteration note the

% indexing utilized from the for loop.

error = error + sqrt((data(i,4) - predict\_pythagreom(i))^2);

end

% this will give us the average error error = error/5

end

# References

Dr. Abhilash Pandya, “Representing Position and Orientation”, Introduction to Robotic Systems I. 10 September 2019.