# ENPM 662: Introduction to Robot Modeling Course Project Report: Kinematic Analysis and MATLAB Simulation of Spherical Wrist Mechanism

Samvrit Srinivas
UID: 114885340

December 31, 2016

# Acknowledgments

I would like to thank the University of Maryland and Professor William Levine for providing an opportunity to work on this project. I would like to thank MathWorks for their incredible online documentation on MATLAB and Simulink which enables even beginners to quickly grasp concepts. I also thank my friends and advisors for their technical support whenever I was stuck.

*- Samvrit Srinivas*

# Contents

# List of Figures



# List of Tables

# 1 Introduction

The tool at the end of a manipulator, also known as the "end-effector", is an important part of a robot, because it is the part of the manipulator that comes directly in contact with the subject of manipulation, and does some meaningful work on the subject.

Most of the industrial robots separate out the positioning and orientation problem. This technique, called decoupling, makes computing the joint angles simpler. Hence, it would be useful to have a computer program that identifies the joint angles that orient the end-effector in a desired manner; and one that can show the orientation of the end-effector, given some combination of joint angles.

# 2 Objective

The objective of this project was to model a simplified spherical wrist with three degrees of freedom[1] using SolidWorks and have the CAD model imported into Simscape for performing simulations of forward and inverse kinematics[2], and to make the robot follow a specified trajectory. It was also stated that the workspace and the configuration space of the spherical wrist would be determined.

# 3 Outcome

A solid model of the spherical wrist was developed using SolidWorks, and the model was imported into Simulink using Simscape Multibody Link. Two graphical user interface (GUI) programs were developed using MATLAB Graphical User Interface Development Environment (GUIDE), one for demonstrating forward kinematics, and the other for inverse kinematics. The simulation of the robot traversing a path from its initial configuration (joint angles equal to zero), to the target joint angles, which were computed using the aforementioned programs, were performed. The workspace of the robot was also computed analytically.

In the forward kinematics (FK) program, the user inputs the joint angles of the robot, and the program computes the resulting orientation of the end-effector.

[1] The number of independent parameters required to define a robot's configuration.

[2] Forward Kinematics: Task of finding the position & orientation (pose) of the end-effector, given the joint angles of the robot. Inverse Kinematics: Task of finding the joint angles of the robot, given the pose of the end-effector.

In the inverse kinematics (IK) program, the user inputs the desired orientation of the robot in terms of a rotation matrix, and the program computes the joint angles which orient the robot in the desired manner.

# 4 Tools Used

## 4.1 SolidWorks

SolidWorks is a solid modeling computer-aided design (CAD) and computer-aided engineering (CAE) computer program that runs on Microsoft Windows. SolidWorks is published by Dassault Systemes [2].

### 4.1.1 Simscape Multibody Link

Simscape Multibody Link is a CAD plug-in for exporting CAD assemblies from SolidWorks, Autodesk Inventor, and PTC Creo software to an XML file detailing the structure and properties of the CAD assembly and 3-D geometry files for visualizing the various CAD parts. The files can then be imported into Simscape Multibody software, which parses the XML data and automatically generates an equivalent multibody model [7].

## 4.2 MATLAB

MATLAB (matrix laboratory) is a multi-paradigm numerical computing environment and fourth-generation programming language. A proprietary programming language developed by MathWorks, MATLAB allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages, including C, C++, C#, Java, Fortran and Python [3].

### 4.2.1 GUI Development Environment (GUIDE)

GUI Development Environment (GUIDE) is a MATLAB tool that allows a programmer to develop a graphical user interface (GUI), in order to simplify the interaction between the end-user and the program. A new GUIDE program creates two files: one that stores the contents and layout of the GUI, and another that contains the MATLAB functions that run when the user interacts with the GUI (each element in the GUI corresponds to a function in the MATLAB file which is

called when there is a change in that particular element). GUIDE documentation has been referenced [6].

### 4.2.2 Simulink

Simulink, developed by MathWorks, is a graphical programming environment for modeling, simulating and analyzing multidomain dynamic systems [4]. A Simulink diagram typically consists of several blocks connected together, which as a whore represents a system model. These blocks have parameters which can be modified make the model more accurate and realistic. These parameters can be set manually, or can be modified through a MATLAB script/program.

### 4.2.3 Robotics Toolbox

This is a MATLAB toolbox developed by Peter Corke, which has provided many functions that are useful for the study and simulation of classical arm-type robotics, for example such things as kinematics, dynamics, and trajectory generation [5].

# 5 Overview of Approach

In the current robotics industry, there is a huge push towards making the human interface to controlling robots easier and more intuitive. Several of the major robotics companies such as ABB and Bosch are investing a lot of resources and effort in this direction, the reason being that the current and the next generation of semi-skilled workers may have lesser technical knowledge and patience to get a hold on complex user-interfaces. Hence, it seems relevant to learn not only modeling of a robot, but also to design simple and intuitive GUI programs to control the robot model.

The first step in this project was to develop a 3-D model of the robot components using SolidWorks. The model is made up of four parts:

1. Base link (fixed)
2. First link (which rotates relative to the base link about the vertical axis)
3. Second link (which rotates relative to the first link about a horizontal axis)
4. Gripper link (which rotates relative to the second link, and which consists of the gripping mechanism)

The second step was to assemble these components using appropriate mates in SolidWorks. The links (components) were aligned appropriately in order for the configuration to comply with the Denavit-Hartenberg convention.

The third step was to export the robot assembly from SolidWorks into MATLAB-readable format using Simscape Multibody Link plugin. The plugin exports the SolidWorks component files into STEP file types, and generates an XML file that contains the mechanical properties of individual components and the spatial description of how the components are assembled together.

The fourth step was to import the model into Simulink through a MATLAB command, which uses the XML file and the STEP files to generate a Simulink block diagram consisting of the solids and revolute joints. Once the diagram was created, slight modifications were done to the revolute joints, and additional blocks were added, in order to actuate them. These additional blocks' parameters specified the trajectory of motion of each revolute joint, and hence were controlled using the GUI programs.

In this project, the user interface has been designed in MATLAB, using the GUIDE tool, which comes built-in. Using this, two programs have been developed: one to analyze the forward kinematics of the robot, and the other to analyze the inverse kinematics. Both these programs drive the same Simulink model which simulates the trajectory of the robot from its initial configuration to its final configuration, which is specified using the GUI. The correctness of the forward kinematics program was verified using Peter Corke's Robotics Toolbox for MATLAB [5].

The workspace and the configuration space of the spherical wrist mechanism was analytically derived.

# 6 Forward Kinematics

In the forward kinematics program, the user inputs the three joint angles of the spherical wrist, and the program computes the orientation of the end-effector in terms of rotations about the standard axes.

Figure 1 shows the GUI for the forward kinematics program, and Figure 2 shows the corresponding finite state-machine diagram. In the forward kinematics GUI, once the user inputs the three joint angles of the spherical wrist and presses the "Create Matrix!" button, the MATLAB program associated with the GUI calculates the rotation part of the transformation matrix, by computing the

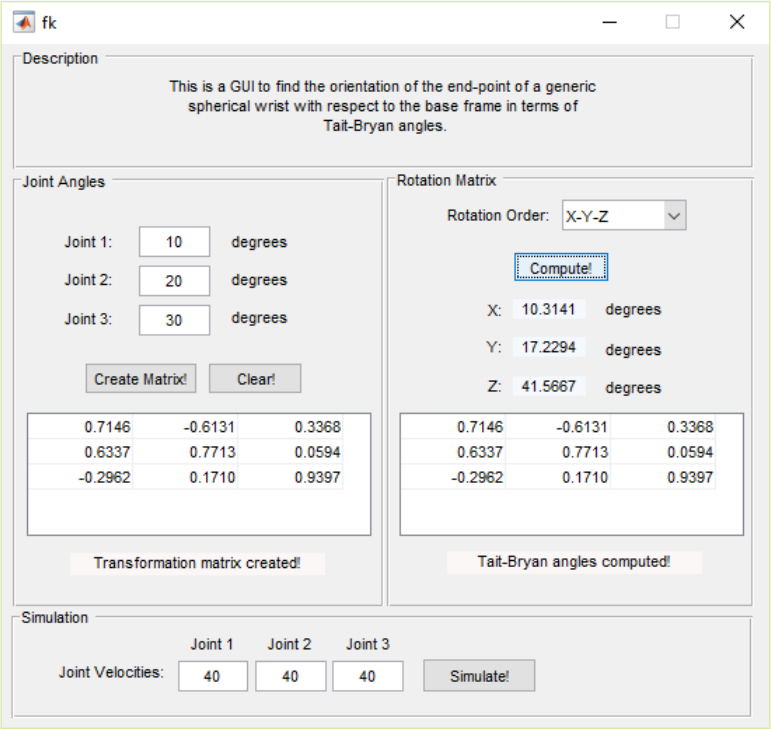


Figure 1: GUI of Forward Kinematics Program

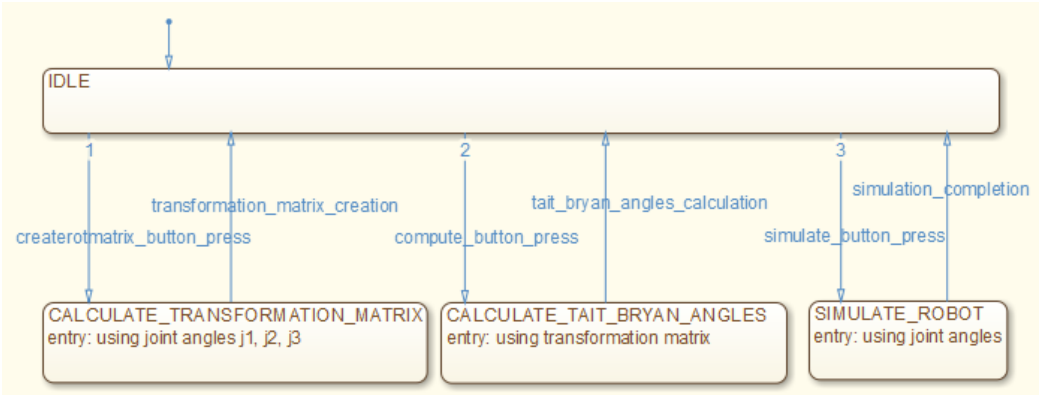


Figure 2: Finite State-Machine Diagram of Forward Kinematics Program

following :

$$T_r = \begin{bmatrix} c_1c_2c_3 - s_1s_3 & -c_1c_2s_3 - s_1c_3 & c_1s_2 \\ s_1c_2c_3 + c_1s_3 & -s_1c_2s_3 + c_1c_3 & s_1s_2 \\ -s_2c_3 & s_2s_3 & c_2 \end{bmatrix} \tag{1}$$

where $c_i, s_i$ are short forms for $\cos(\theta_i)$ and $\sin(\theta_i)$ respectively, and $\theta_i, i = 1, 2, 3$ are the three joint angles of the spherical wrist. The above transformation matrix is obtained by multiplying the individual link transformation matrices, which are a function of the D-H parameters of the links [1]. The goal of the forward kinematics program is to find the orientation of the end-effector in terms of the Tait-Bryan angles[3]. In the GUI, the user can select the desired order of rotation in which the orientation needs to be expressed. The order of rotation is important, because it influences the resulting rotation matrix, as matrix multiplication is not commutative. For example, if the user chooses the order X-Y-Z (where the resulting frame is first rotated with respect to the fixed frame about the x-axis, followed by a rotation about the new y-axis and then followed by a rotation about the newer z-axis). The goal is to find the extent to which the rotations have taken place in each of the axes. Let $x, y, z$ be the extent of rotation about each of the axes. Then the resulting rotation matrix $R$ can be written as:

$$\begin{aligned} R &= Rotz(z) * Roty(y) * Rotx(x) \\ &= \begin{bmatrix} c_zc_y & -s_zc_x + c_zs_ys_x & s_zs_x + c_zs_x + c_zs_yc_x \\ s_zc_y & c_zc_x + s_zs_ys_x & -c_zs_x + s_zs_yc_x \\ -s_y & c_ys_x & c_yc_x \end{bmatrix} \end{aligned} \tag{2}$$

where $Rotx, Roty, Rotz$ are the rotation matrices corresponding to rotations about the x-axis, y-axis and the z-axis, through the angles $x, y, z$ respectively.

We know that the matrices $R$ and $T_r$ must be equal. Hence, comparing the elements of matrix $T_r$, which consists of numerical values, to the corresponding elements of $R$, which are functions of $x, y, z$, we can find the values of $x, y, z$ that satisfy the equations. If $t_{31}, t_{33}, t_{11}$ are the $t_{ij}{}^{\text{th}}$ elements of matrix $T_r$, then

[3]Tait-Bryan angles are a subset of Euler angles, in which there is one rotation about each of the standard axes.

we have:

$$y = \arcsin\left(-t_{31}\right) \tag{3}$$

$$x = \arccos\left(\frac{t_{33}}{\cos(y)}\right) \tag{4}$$

$$z = \arccos\left(\frac{t_{11}}{\cos(y)}\right) \tag{5}$$

The above computation is done when the "Compute!" button is pressed, and the $x, y, z$ rotations in degrees are displayed. In order to verify if the computation is correct, the program also evaluates matrix $R$ using Equation 2, to compute the rotation matrix, which should be the same as the transformation matrix $T_r$.

In the similar manner, we can find the values of $x, y, z$ for any order of rotation. For different rotation orders, different resultant rotation matrices are formed and we need to appropriately choose the elements of $T_r$ and $R$ to compare (see "tbeval.m" in Appendix D).

The results of the forward kinematics program was verified using Peter Corke's Robotics Toolbox for MATLAB. The toolbox allows one to model a robot using its D-H parameters and simulate it. The simulation allows the user to set the joint angles, and it displays the orientation and translation of the end-effector frame with respect to the base frame. In Figure 3, we can see that for the same joint angles as input, we get the same orientation in terms of Tait-Bryan angles (the values in the corresponding green and red boxes match).

# 7 Inverse Kinematics

In the inverse kinematics program, the user inputs the desired orientation in terms of Tait-Bryan angles and the GUI computes the three joint angles of the spherical wrist that together produce the desired orientation.

Figure 4 shows the GUI for the inverse kinematics program. On the left part of the GUI, the user inputs the order of rotation, and the extent of rotation in degrees about each of the standard axis. When the "Create Matrix!" button is pressed, the MATLAB program associated with the GUI computes the corresponding rotation matrix for the given order of rotation. For example, the rotation matrix $R$ (see Section 6) for the order X-Y-Z is given in Equation 2. Also, the rotation part of the homogeneous transformation matrix $T_r$ (see Section 6) is obtained by multiplying the individual link transformation matrices, which are a function of the D-H parameters of the links [1], and is shown in Equation 1.

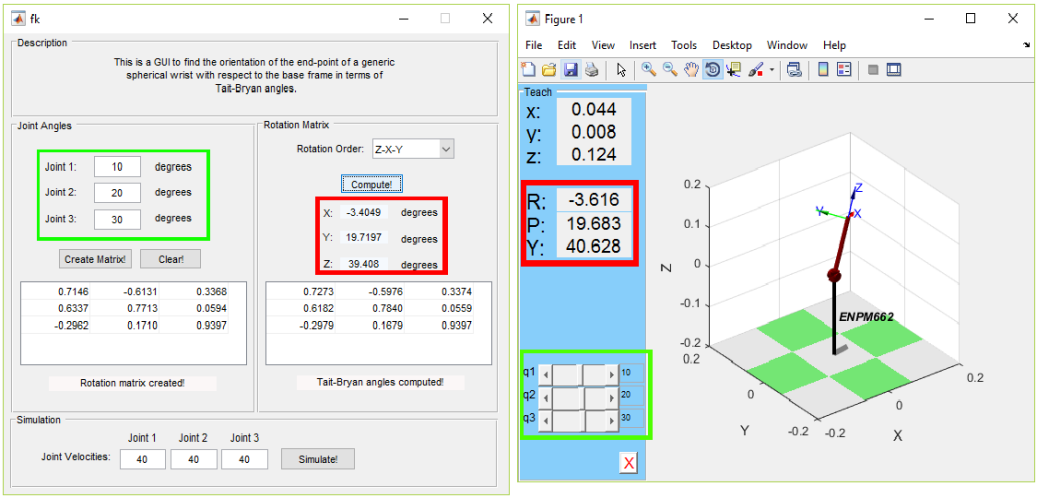


Figure 3: MATLAB Robotics Toolbox

In the inverse kinematics problem, matrix $R$ consists of numerical values, which we compare with the corresponding elements of $T_r$, which contain the joint variables. Appropriately choosing the matrix elements to compare, allows us to evaluate the joint angles of the spherical wrist that results in the desired orientation. If $r_{33}, r_{23}, r_{32}$ are the $r_{ij}{}^{\text{th}}$ elements of matrix $R$, and $j_1, j_2, j_3$ are the three joint angles of the spherical wrist, then we have:

$$j_2 = \arccos\left(r_{33}\right) \tag{6}$$

$$j_1 = \arcsin\left(\frac{r_{23}}{\sin(j_2)}\right) \tag{7}$$

$$j_3 = \arcsin\left(\frac{r_{32}}{\sin(j_2)}\right) \tag{8}$$

The above computation is done when the "Compute Joint Angles!" button is pressed. Again, in order to verify if the resulting orientation is correct, the rotation part of the homogeneous transformation matrix $T_r$ is constructed using Equation 1 to see if it is the same as matrix $R$ computed using the Tait-Bryan angles.

Similarly, the corresponding joint angles can be found for any order of rotation, by comparing appropriate elements of matrices $R$ and $T_r$.

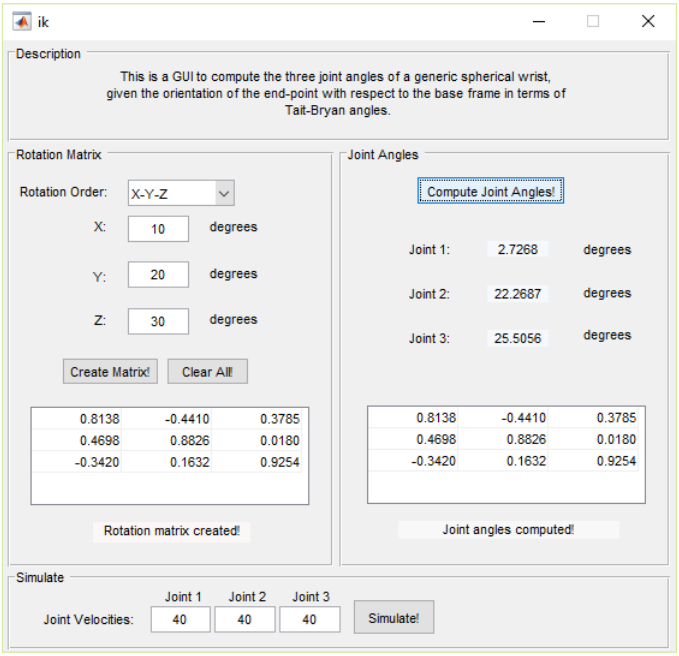


Figure 4: GUI of Inverse Kinematics Program

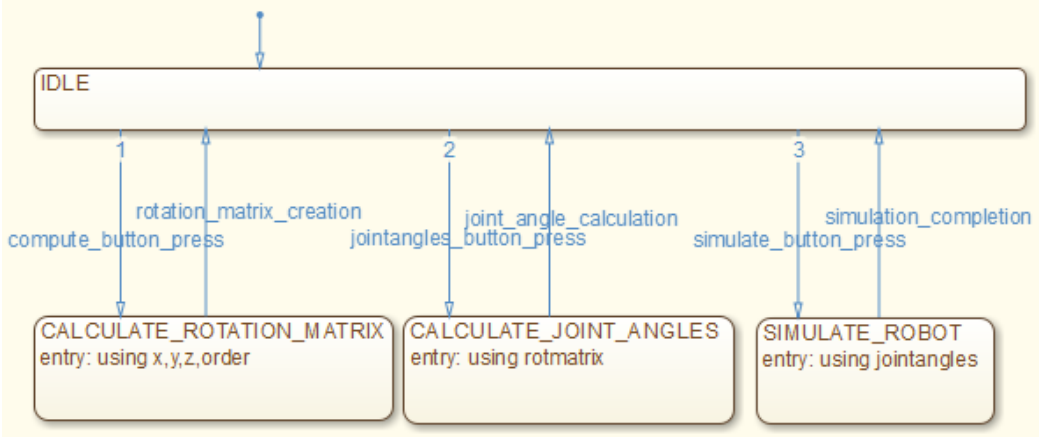


Figure 5: Finite State-Machine Diagram of Inverse Kinematics Program

# 8 Simulink Simulation

The Simulink model is obtained by importing the files that are exported by Simulink Multibody Link from SolidWorks (component .STEP files and the .XML file). This creates a Simulink block diagram that consists of the following types of blocks, along with the configuration blocks:

1. Solid
2. Rigid Transformation
3. Revolute

Each joint in the robot model corresponds to a "Revolute" block in the Simulink diagram that facilitates relative motion between the preceding and succeeding links. The links, on the other hand are represented by the "Solid" blocks. Hence there are totally three "Revolute" and four "Solid" blocks in the Simulink diagram. By default, the revolute joints are not actuated. In order for the robot model to follow a path from its initial configuration/orientation to its desired orientation, an appropriate input signal has to be provided to the revolute joints for either torque or position (the other is automatically calculated).

In this simulation, input is given to the positions of the revolute joints at discrete time, and the torque is computed automatically. The input to the joint position is supplied by a "Ramp" block, which provides a ramp signal, with zero as the initial value and increasing linearly at a slope of $40$. At any time instant, the joint angle is equal to the value provided by the ramp input at that time instant. The slope is nothing but the joint velocity, and hence has to be set to the required joint velocity in deg/sec by the MATLAB program. The ramp input does not have a final value, and the joint keeps rotating indefinitely unless another block is added in series which saturates the input. Thus, if the joints are to stop rotating when the desired joint angle is reached, a "Saturation" block has to be added and the saturation limits must be set to the required joint angle by the MATLAB program. The ramp input, however, does not provide negative output. This can be an issue when the desired joint angle is negative, for example $-30^\circ$. It can be easily corrected by adding a "Gain" block in series, and setting the gain value to be either $+1$ or $-1$ depending on the sign of the desired joint angle.

As mentioned in Section 5, both the forward and inverse kinematics programs run the same Simulink simulation (see Appendix C for Simulink block diagram). The simulation requires the input of the three desired joint angles in degrees and

their corresponding angular velocities in deg/sec. The simulation begins with the robot model in initial configuration (all the three joint angles equal to zero degrees). After an interval of 2 seconds, each joint starts rotating at a constant speed (specified by the user) until it reaches the desired joint angle (specified by the user in the forward kinematics program; computed by the MATLAB program in the inverse kinematics program).

Once the "Simulate!" button is pressed in either of the GUIs, the MATLAB program associated with the GUI does the following:

1. Retrieve the joint angles from GUI
2. Retrieve joint angular velocities from GUI
3. Open the Simulink file
4. Set or modify the appropriate block parameters
5. Run the Simulink file
6. Save the Simulink file

Figure 7 shows the finite state-machine diagram of the simulation process. After the simulation, we can see that the Scope shows the evolution of the joint angles from the robot's initial configuration to its final configuration.

# 9 Workspace and Configuration Space

The workspace of the robot can be defined as the total volume swept by the end-effector of the robot as it executes all possible motions. The configuration space on the other hand is the set of all configurations of the robot.

| i | $a_i$ | $\alpha_i$ | $d_i$ | $\theta_i$ |
|---|---|---|---|---|
| 1 | 0 | $-90$ | 0 | $\theta_1$ |
| 2 | 0 | 90 | 0 | $\theta_2$ |
| 3 | 0 | 0 | $d = 132\,\text{mm}$ | $\theta_3$ |

Table 1: Denavit-Hartenberg table for spherical wrist mechanism [1]

Figure 8 shows the spherical wrist model in D-H conventions, and the D-H table is shown in Table 1. The complete homogeneous transformation matrix is

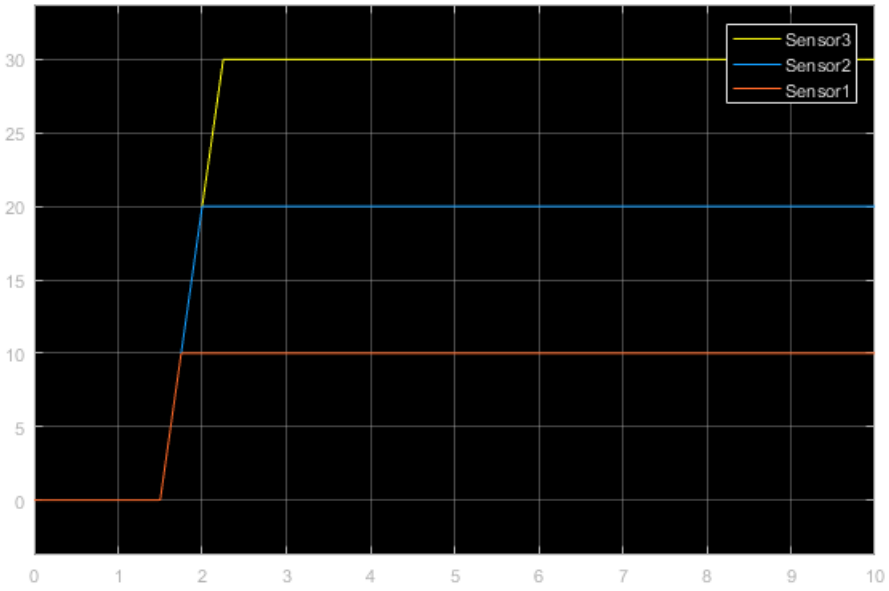


Figure 6: Simulink Scope reading

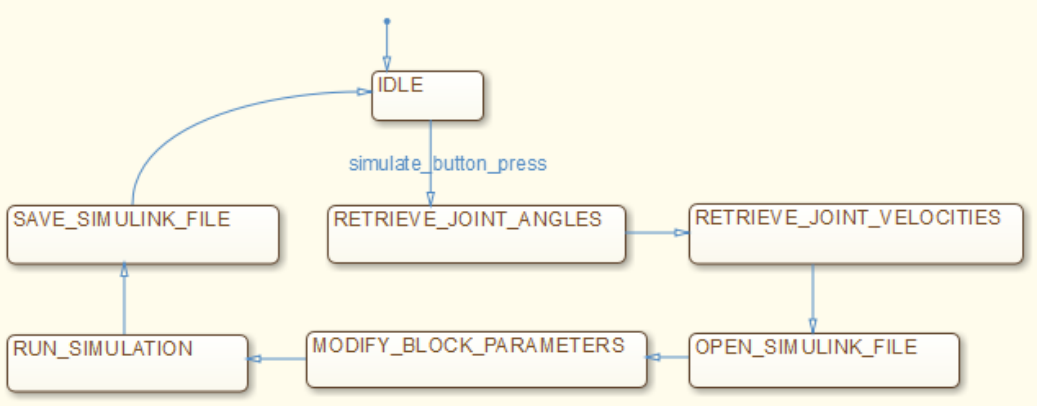


Figure 7: Finite State-Machine Diagram of Simulation Process

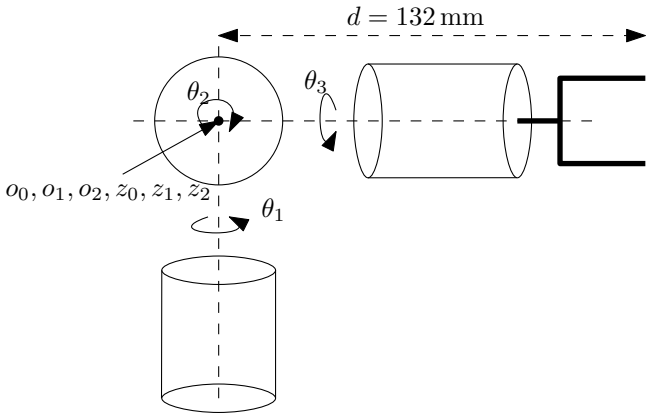


Figure 8: Spherical wrist in D-H conventions

as follows [1]

$$T = \begin{bmatrix} c_1c_2c_3 - s_1s_3 & -c_1c_2s_3 - s_1c_3 & c_1s_2 & c_1s_2d \\ s_1c_2c_3 + c_1s_3 & -s_1c_2s_3 + c_1c_3 & s_1s_2 & s_1s_2d \\ -s_2c_3 & s_2s_3 & c_2 & c_2d \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9}$$

The workspace of the robot can be determined by analyzing the first three elements of the fourth column of $T$, which represent the coordinates of the end-effector with respect to the base frame. Squaring and adding the terms, we get:

$$\begin{aligned} x^2 + y^2 + z^2 &= {c_1}^2{s_2}^2d^2 + {s_1}^2{s_2}^2d^2 + {c_2}^2d^2 \\ &= d^2\left[{s_2}^2\left({c_1}^2 + {s_1}^2\right) + {c_2}^2\right] \\ &= d^2\left({s_2}^2 + {c_2}^2\right) \\ &= d^2 \end{aligned}$$

From the above computation, we get $x^2 + y^2 + z^2 = d^2$, which is an equation for a sphere with radius $d = 132\,\text{mm}$. This is the rationale behind the name "spherical wrist". However, in practice, a spherical wrist will be attached to a robotic arm, and hence some constraints will be imposed on the limits of rotation of the joints. Hence, assuming that joint 1 can rotate $0° - 360°$ and joint 2 can rotate $0° - 180°$, we can state that the workspace of the spherical wrist is the surface of a hemisphere. Hence, the end-effector of the spherical wrist can reach any point on the surface of the hemisphere with radius $d$.

The configuration space of the spherical wrist is the set of all configurations of the robot. Since the end-effector traces the surface of a hemisphere, we know that the body of the spherical wrist mechanism sweeps the entire volume of the same hemisphere. Hence, the configuration space is the set of all points within and on the hemisphere.

# 10 Conclusion

The forward and inverse kinematics have been analyzed for a spherical wrist robot, and a SolidWorks CAD model has been imported into Simulink for simulation of the robot. The workspace and configuration space of the robot has also been determined. As the computation of the forward and inverse kinematics is independent of the 3-D model, the GUIs can be used to analyze the kinematics of any spherical wrist as long as it has the same D-H parameters. The simulation too can be adapted into any spherical wrist Simscape model with slight modifications to the Simulink blocks.

The key takeaways from this project were:

1. The kinematic analysis of a spherical wrist mechanism.
2. The skill of designing interactive GUIs using MATLAB.
3. The concept of programmatically modifying Simulink block parameters and running simulations through a MATLAB script.

Following is a (non-exhaustive) list of some of the specific topics that have been learned in the process of completing this project, even though some of them were not used in the project itself:

## 10.1 Learning

### 10.1.1 SolidWorks

1. **Hinge Mate**: This is a type of mate that is used to define a one DOF joint in which one body rotates with respect to another. One pair of concentric and one pair of coincident surfaces are to be specified to define this type of mate.
2. **Symmetric Mate**: This is a type of mate that defines the motion of two bodies to be symmetric about a plane.

### 10.1.2 MATLAB

1. **GUIDE & App Designer**: These are two tools in MATLAB that can be used to create GUIs. Both have pros and cons, with App Designer being more aesthetic but with less flexibility than GUIDE.

2. **Stateflow**: This is a MATLAB package like Simulink that allows one to develop finite state-machine diagrams.

3. **Commands**: Some of the MATLAB commands that were newly understood were:

   (a) `smimport()`: This command imports the XML file generated by Simscape Multibody Link, into a Simulink block diagram.

   (b) `set_param()`: This is a command that allows one to set/modify Simulink block parameters through the MATLAB command line/script.

   (c) `open_system()`: This is a command that opens a Simulink file, subsystem or block dialogue box.

   (d) `load_system()`: This is a command that invisibly loads a Simulink model.

   (e) `save_system()`: This is a command that saves a Simulink model, subsystem or block dialogue box.

   (f) `close_system()`: This is a command that closes a Simulink model, subsystem or block dialogue box.

   (g) `rotx(),roty(),rotz()`: These functions provide the corresponding rotation matrices for specified angles (expressed in radians).

4. **Simulink Blocks**: The second generation Simulink library was explored and the differences from the first generation library were learned. Some of the Simulink blocks that were newly understood were:

   (a) *Constant*: This is a block that provides a constant signal.

   (b) *Random*: This block provides a random signal with a mean value.

   (c) *Ramp*: This block gives a ramp input, starting from an initial value, increasing at a constant rate defined by the slope.

   (d) *Saturation*: This is a block that is used to constrain a signal by adding upper and lower limits to its value.

(e) *Gain*: This is a block that multiplies a signal by a constant, that is defined by the gain value.

## 10.2 Future Work

The 3-D model of this spherical wrist has been designed to fit as an end-effector into a low-cost robotic arm that was developed as a previous undergraduate project (shown by Figure 9) at RNS Institute of Technology and later at the Indian Institute of Science [8]. With slight modifications, this end-effector can be manufactured and attached to the robotic arm, and can be used as an educational tool. Future work can include:

1. Development of dynamics equations for the entire robot.
2. Development of path planning and obstacle detection algorithms.
3. Development of the human-robot interface for controlling the robot.
4. Development of pick and place algorithms using computer vision.

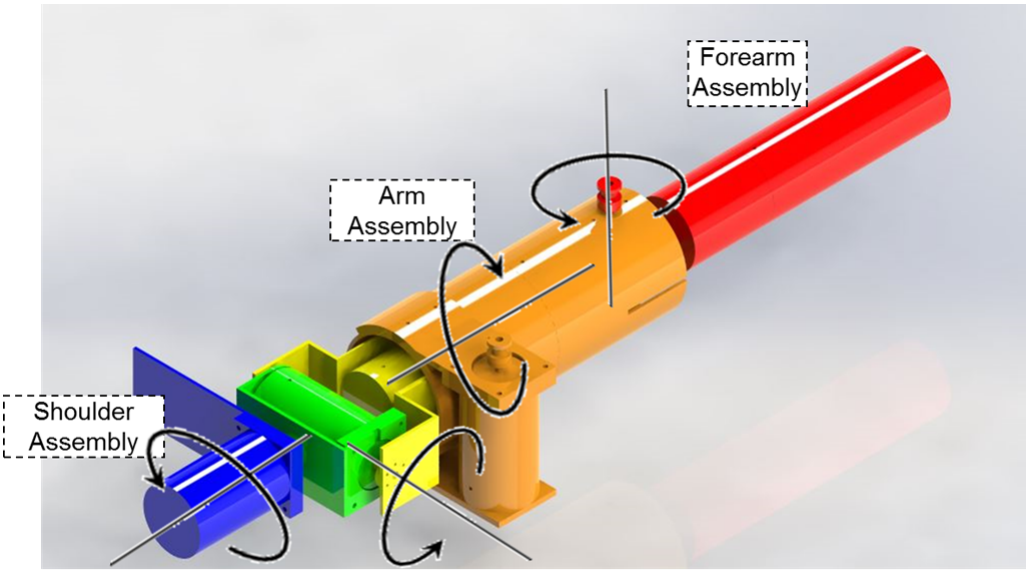


Figure 9: RNSIT Robot

# A Appendix I: Instructions on How to Run Simulation

Video tutorial: *https://youtu.be/O7YBvMSepTg*

## A.1 Setup

1. Extract "Simscape.zip" and ensure that all the extracted files from the .zip file are within a single folder.
2. Open MATLAB and set working directory to the folder referenced in the previous step.
3. In the MATLAB command window, type `fk` for the forward kinematics program, and `ik` for the inverse kinematics program, and press return.

## A.2 Forward Kinematics Program (fk)

1. In the "Joint Angles" panel, type in the three joint angles of the spherical wrist in degrees and press "Create Matrix!" button. The "Clear!" button can be used to clear the input and the transformation matrix.
2. In the "Rotation Matrix" panel, choose the desired rotation order in which the orientation is to be expressed, and then click on "Compute!".
3. In the "Simulation" panel, input the desired joint velocities in deg/sec and click on "Simulate!".

## A.3 Inverse Kinematics Program (ik)

1. In the "Rotation Matrix" panel, choose the rotation order and input the Tait-Bryan angles. The "Clear!" button can be used to clear the input and the rotation matrix.
2. In the "Joint Angles" panel, click on "Compute Joint Angles!" button.
3. In the "Simulation" panel, input the desired joint velocities in deg/sec and click on "Simulate!".

Note:- If it is the first time "Simulate!" button is pressed since MATLAB startup, it takes a long time for Simulink to open. This is normal.

# B Appendix II: CAD Model

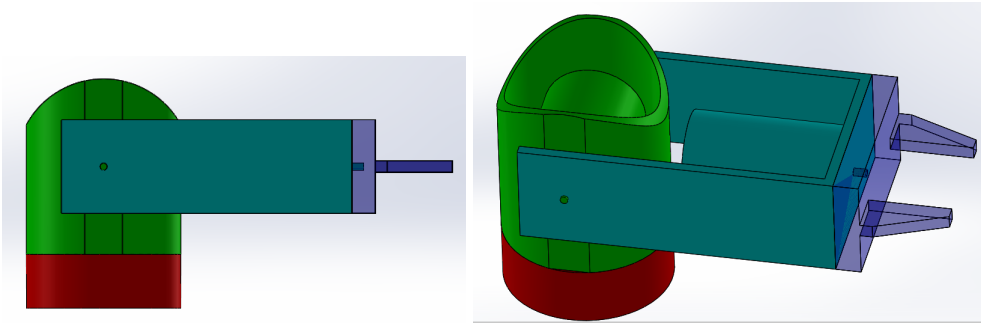

Figure 10: SolidWorks CAD model

## B.1 Specs

The following data has been obtained from the file generated during the import of the XML file into Simulink.

### B.1.1 Part No. 1

1. Material: 1060 Aluminum Alloy
2. Mass = 0.05987 kg
3. CoM = [0 15.34 0] mm
4. MoI = [26.55 46.97 26.55] kg*mm$^2$

### B.1.2 Part No. 2

1. Material: Delrin 2700 NC010, Low Viscosity, Acetal Copolymer
2. Mass = 0.03481 kg
3. CoM = [7.14 0 36] mm
4. MoI = [20.23 17.90 5.90] kg*mm$^2$

### B.1.3 Part No. 3

1. Material: 1060 Aluminum Alloy
2. Mass = 0.29058 kg
3. CoM = [82.77 20.00 0] mm
4. MoI = [175.09 425.71 310.43] kg*mm$^2$

### B.1.4 Part No. 4

1. Material: ABS Plastic
2. Mass = 0.16480 kg
3. CoM = [-0.00039 28.38 -0.00035] mm
4. MoI = [103.45 88.05 99.66] kg*mm$^2$

## B.2 Remarks

1. The appropriate way to define revolute joints in SolidWorks is to use the "Hinge" mate, however, this is not recognized by Simscape Multibody Link, and hence a combination of "Concentric" and "Coincident" mates are used.
2. The materials chosen are not completely arbitrary. For example, the base part requires higher rigidity than the gripper portion, which needs some flexibility. Hence Aluminum with low elastic modulus is chosen for the base part and ABS plastic with higher elastic modulus is chosen for the gripper portion. The manufacturability has also been taken into consideration while choosing the materials.

# C Appendix III: Simulink Diagram

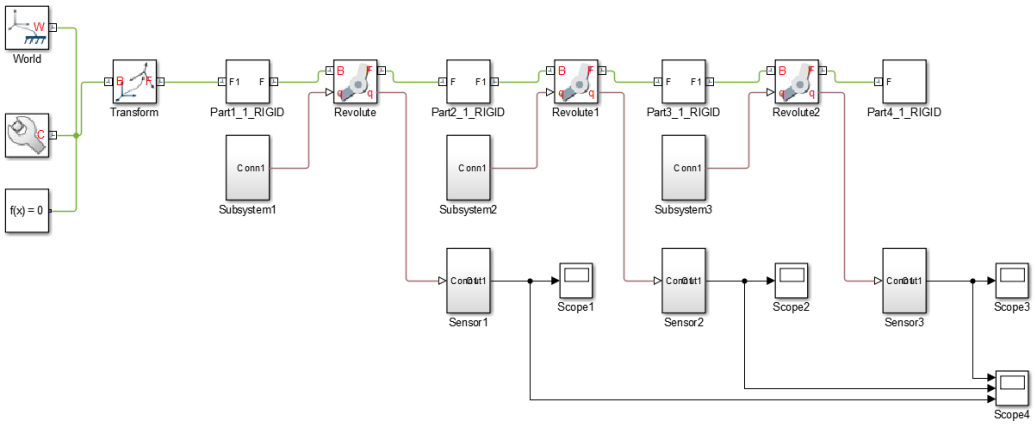


Figure 11: Simulink block diagram

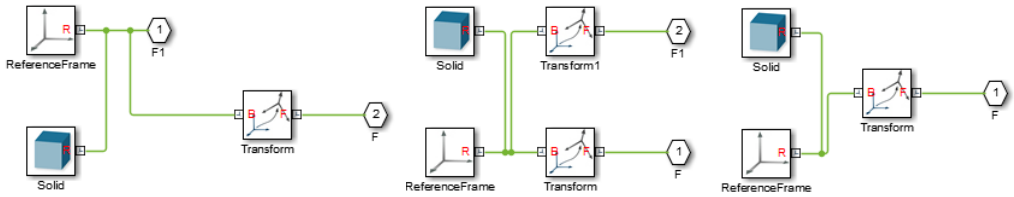


Figure 12: Part diagrams

## C.1 Remarks

1. Since the ramp input is time-varying, it is required to supply the signals for the velocity and acceleration along with the position. This is done using the "Derivative" blocks.

2. In the "Simulink-PS Converter" block, under "Input Handling" tab, the "Filtering and derivatives" parameter must be set to "Provide input derivative(s)", and the "Input derivatives" parameter must be set to "Provide first and second derivatives".

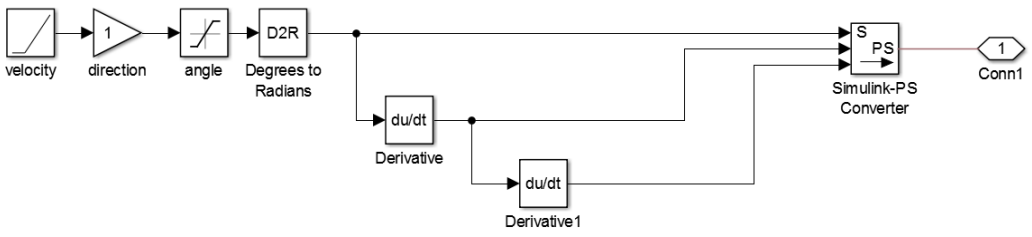


Figure 13: Actuation subsystem diagram

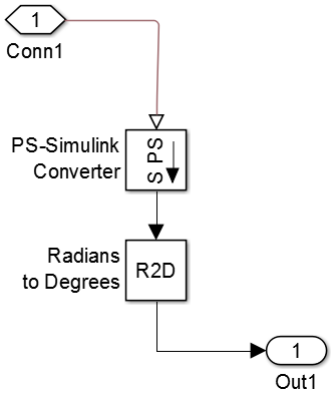


Figure 14: Sensor subsystem diagram

3. In the actuation subsystem, the “Ramp” block is renamed to “velocity”, “Gain” to “direction” and “Saturation” to “angle”, based on their functions. The names of these blocks are to be noted, as they will be referenced in the MATLAB program while setting/modifying their parameters.

# D Appendix IV: MATLAB Programs

Since the entire MATLAB programs are too lengthy to be included in the report, only the snippets that are functional will be shown.

Each element in the GUI has a MATLAB function tied to it that is executed when there is a change in that element. Hence, the functions that are tied to the different buttons in the GUI are shown below (in addition to the function performed at GUI startup). Each element is identified by a unique "Tag" name (eg., `rotmatrix, j1, transmatrix, j1vel`, etc.), using which, the value contained that element can be retrieved or set.

## D.1 Forward Kinematics Program (fk.m)

This is the MATLAB program that is executed when the user interacts with the forward kinematics GUI.

```
% --- Executes just before fk is made visible.
function fk_OpeningFcn(hObject, eventdata, handles, varargin)
% Choose default command line output for fk
handles.output = hObject;
% Update handles structure
guidata(hObject, handles);
% set both matrices to identity matrix
rotmatrix = [1 0 0 ; 0 1 0 ; 0 0 1];
set(handles.rotmatrix,'data',rotmatrix);
set(handles.rotmatrix1,'data',rotmatrix);

% --- Executes on button press in createtransmatrix.
function createtransmatrix_Callback(hObject, eventdata, handles)
% get joint angles from user input
j1 = str2double(get(handles.j1,'string'));
j2 = str2double(get(handles.j2,'string'));
j3 = str2double(get(handles.j3,'string'));
% evaluate transformation matrix
transmatrix = evaltransmatrix(j1,j2,j3);
% display transformation matrix
set(handles.transmatrix,'data',transmatrix);
% display result of operation
set(handles.rotmatrixresult,'string','Transformation matrix
    created!');

% --- Executes on button press in clear.
function clear_Callback(hObject, eventdata, handles)
```

```
% makes all inputs zero
set(handles.j1,'string','0');
set(handles.j2,'string','0');
set(handles.j3,'string','0');
transmatrix = [1 0 0 ; 0 1 0 ; 0 0 1];
set(handles.transmatrix,'data',transmatrix);

% --- Executes on button press in simulate.
function simulate_Callback(hObject, eventdata, handles)
j1 = get(handles.j1,'string');
j2 = get(handles.j2,'string');
j3 = get(handles.j3,'string');
% get joint velocities from user input
j1vel = get(handles.j1vel,'string');
j2vel = get(handles.j2vel,'string');
j3vel = get(handles.j3vel,'string');
% execute simulation
execute(j1,j2,j3,j1vel,j2vel,j3vel);
guidata(hObject,handles);
```

## D.2 Inverse Kinematics Program (ik.m)

This is the MATLAB program that is executed when the user interacts with the inverse kinematics GUI.

```
% --- Executes just before fk is made visible.
function fk_OpeningFcn(hObject, eventdata, handles, varargin)
% Choose default command line output for fk
handles.output = hObject;
% Update handles structure
guidata(hObject, handles);
% set both matrices to identity matrix
rotmatrix = [1 0 0 ; 0 1 0 ; 0 0 1];
set(handles.rotmatrix,'data',rotmatrix);
set(handles.rotmatrix1,'data',rotmatrix);

% --- Executes on button press in rotmatrix.
function rotmatrix_Callback(hObject, eventdata, handles)
% get the tait-bryan angles from user input
x = str2double(get(handles.editx,'string'));
y = str2double(get(handles.edity,'string'));
z = str2double(get(handles.editz,'string'));
% get the order of rotation from user input
order = num2str(get(handles.rotorder,'value'));
% calculate rotation matrix
```

```
rotmatrix = calcrotmatrix(x,y,z,order);
% display rotation matrix
set(handles.matrix,'data',rotmatrix);
% display result of operation
set(handles.rotmatresult,'string','Rotation matrix created!');

% --- Executes on button press in jointangles.
function jointangles_Callback(hObject, eventdata, handles)
% get tait-bryan angles from user input
x = str2double(get(handles.editx,'string'));
y = str2double(get(handles.edity,'string'));
z = str2double(get(handles.editz,'string'));
% get order of rotation from user input
order = num2str(get(handles.rotorder,'value'));
% calculate rotation matrix
rotmatrix = calcrotmatrix(x,y,z,order);
% compute joint angles using rotation matrix
[j1,j2,j3] = calcjointangles(rotmatrix);
% calculate transformation matrix using joint angles
tramsmatrix = evaltransmatrix(j1,j2,j3);
% display joint angles
set(handles.joint1,'string',num2str(j1));
set(handles.joint2,'string',num2str(j2));
set(handles.joint3,'string',num2str(j3));
% display transformation matrix
set(handles.transmatrix,'data',tramsmatrix);
% check if joint angles exist, and display result
if((strcmp(num2str(j1),'NaN') == 0) && (strcmp(num2str(j2),'NaN'
    ) == 0) && (strcmp(num2str(j3),'NaN') == 0))
     set(handles.result,'string','Joint angles computed!');
else
     set(handles.result,'string','Joint angles could not be
    computed!');
end

% --- Executes on button press in simulate.
function simulate_Callback(hObject, eventdata, handles)
% get joint angles from GUI data
j1 = get(handles.joint1,'string');
j2 = get(handles.joint2,'string');
j3 = get(handles.joint3,'string');
% get joint velocities from user input
j1vel = get(handles.j1vel,'string');
j2vel = get(handles.j2vel,'string');
j3vel = get(handles.j3vel,'string');
```

```
% execute simulation
execute(j1,j2,j3,j1vel,j2vel,j3vel);
guidata(hObject,handles);
```

## D.3 Transformation Matrix (evaltransmatrix.m)

This user-defined function, called by the forward kinematics program, computes the rotation part of the transformation matrix using the three joint angles of the spherical wrist.

```
function transmatrix = evaltransmatrix(a,b,c)
% compute all the elements of the rotation part of
    transformation matrix
r11 = (cosd(a)*cosd(b)*cosd(c)) - (sind(a)*sind(c));
r12 = - (cosd(a)*cosd(b)*sind(c)) - (sind(a)*cosd(c));
r13 = cosd(a)*sind(b);
r21 = (sind(a)*cosd(b)*cosd(c)) + (cosd(a)*sind(c));
r22 = - (sind(a)*cosd(b)*sind(c)) + (cosd(a)*cosd(c));
r23 = sind(a)*sind(b);
r31 = -sind(b)*cosd(c);
r32 = sind(b)*sind(c);
r33 = cosd(b);
% put them all together into a 3x3 matrix
transmatrix = [r11 r12 r13 ; r21 r22 r23 ; r31 r32 r33];
```

## D.4 Rotation Matrix (calcrotmatrix.m)

This user-defined function, called by the inverse kinematics program, computes the rotation matrix corresponding to the order of desired rotations of standard axes.

```
function rotmatrix = calcrotmatrix(x,y,z,order)
% compute rotation matrix
% convert angles from degrees to radians
x = deg2rad(x);
y = deg2rad(y);
z = deg2rad(z);
% calculate rotation matrix based on order of rotation
switch order
    case '1'    % X-Y-Z
        rotmatrix = rotz(z)*roty(y)*rotx(x);
    case '2'    % X-Z-Y
        rotmatrix = roty(y)*rotz(z)*rotx(x);
```

```
    case '3'    % Y–X–Z
        rotmatrix = rotz(z)*rotx(x)*roty(y);
    case '4'    % Y–Z–X
        rotmatrix = rotx(x)*rotz(z)*roty(y);
    case '5'    % Z–Y–X
        rotmatrix = rotx(x)*roty(y)*rotz(z);
    case '6'    % Z–X–Y
        rotmatrix = roty(y)*rotx(x)*rotz(z);
end
```

## D.5 Execute Simulation (execute.m)

This user-defined function, called by both forward and inverse kinematics programs, modifies the Simulink block parameters, runs the simulation and saves the Simulink file with the modified block parameters.

```
function status = execute(j1,j2,j3,j1vel,j2vel,j3vel)
% execute simulation
% set up variables that refer to blocks in Simulink diagram
sys = 'Assem1';
joint1 = strcat(sys,'/Subsystem1/');
joint2 = strcat(sys,'/Subsystem2/');
joint3 = strcat(sys,'/Subsystem3/');
joint1velocity = strcat(joint1,'velocity');
joint2velocity = strcat(joint2,'velocity');
joint3velocity = strcat(joint3,'velocity');
joint1direction = strcat(joint1,'direction');
joint2direction = strcat(joint2,'direction');
joint3direction = strcat(joint3,'direction');
joint1angle = strcat(joint1,'angle');
joint2angle = strcat(joint2,'angle');
joint3angle = strcat(joint3,'angle');

% open Simulink file
open_system(sys);

% if joint velocity field is empty, make it 40
if(isempty(j1vel))
    j1vel = '40';
end
if(isempty(j2vel))
    j2vel = '40';
end
if(isempty(j3vel))
```

```
    j3vel = '40';
end

% set joint velocities by modifying slope of ramp input
set_param(joint1velocity,'slope',j1vel);
set_param(joint2velocity,'slope',j2vel);
set_param(joint3velocity,'slope',j3vel);

% check sign of joint angle and set the limit of Saturation and
    Gain block appropriately
if((strcmp(j1,'NaN')==0)&&(strcmp(j2,'NaN')==0)&&(strcmp(j3,'NaN
    ')==0))
    if(sign(str2double(j1))==1)
        set_param(joint1direction,'Gain','1');
        set_param(joint1angle,'UpperLimit',num2str(j1));
    elseif(sign(str2double(j1))==-1)
        set_param(joint1direction,'Gain','-1');
        set_param(joint1angle,'LowerLimit',num2str(j1));
    else
        set_param(joint1direction,'Gain','1');
        set_param(joint1angle,'UpperLimit',num2str(j1));
    end
    if(sign(str2double(j2))==1)
        set_param(joint2direction,'Gain','1');
        set_param(joint2angle,'UpperLimit',num2str(j2));
    elseif(sign(str2double(j2))==-1)
        set_param(joint2direction,'Gain','-1');
        set_param(joint2angle,'LowerLimit',num2str(j2));
    else
        set_param(joint2direction,'Gain','1');
        set_param(joint2angle,'UpperLimit',num2str(j2));
    end
    if(sign(str2double(j3))==1)
        set_param(joint3direction,'Gain','1');
        set_param(joint3angle,'UpperLimit',num2str(j3));
    elseif(sign(str2double(j3))==-1)
        set_param(joint3direction,'Gain','-1');
        set_param(joint3angle,'LowerLimit',num2str(j3));
    else
        set_param(joint3direction,'Gain','1');
        set_param(joint3angle,'UpperLimit',num2str(j3));
    end
    % run simulation
    sim(sys);
end
```

```
% save Simulink file with modified block parameters
save_system(sys);
status = 1;
```

## D.6 Evaluate Tait-Bryan Angles (tbeval.m)

This user-defined function, called by the forward kinematics program, computes the Tait-Bryan angles by comparing the rotation matrix with the transformation matrix by performing element-by-element comparison.

```
function [x,y,z] = tbeval(rotorder,r)
% evaluate tait-bryan angles depending on desired rotation order
switch rotorder
    case '1'    % X-Y-Z
        y = asind(-r(3,1));
        x = acosd(r(3,3)/cosd(y));
        z = acosd(r(1,1)/cosd(y));
    case '2'    % X-Z-Y
        z = asind(r(2,1));
        x = acosd(r(2,2)/cosd(z));
        y = acosd(r(1,1)/cosd(z));
    case '3'    % Y-X-Z
        x = asind(r(3,2));
        y = acosd(r(3,3)/cosd(x));
        z = acosd(r(2,2)/cosd(x));
    case '4'    % Y-Z-X
        z = asind(-r(1,2));
        y = asind(r(1,3)/cosd(z));
        x = asind(r(3,2)/cosd(z));
    case '5'    % Z-X-Y
        x = asind(-r(2,3));
        y = asind(r(1,3)/cosd(x));
        z = asind(r(2,1)/cosd(x));
    case '6'    % Z-Y-X
        y = asind(r(1,3));
        z = acosd(r(1,1)/cosd(y));
        x = acosd(r(3,3)/cosd(y));
end
```

## D.7 Robotics Toolbox (toolbox.m)

This is a program used to verify the correctness of the forward kinematics program.

```
% ENPM662 - Project
% Program to verify correctness of Simscape Simulations
% Samvrit Srinivas (UID: 114885340)

% define d-h parameters of each link (theta, d, a, alpha)
l1 = Link([0 0 0 -pi/2]);
l2 = Link([0 0 0 pi/2]);
l3 = Link([0 0.132 0 0]);

% build robot using links
r = SerialLink([l1 l2 l3],'name','ENPM662');

% visualise robot
r.teach([0 0 0]);
```

# E Appendix V: GUI Development Environment

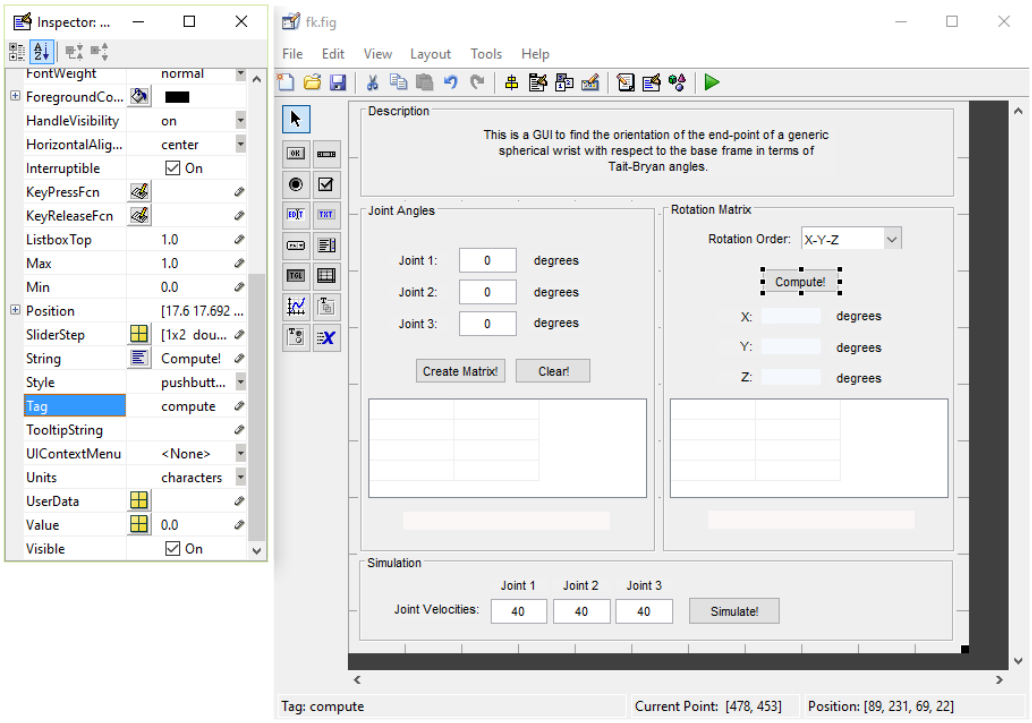


Figure 15: GUI of forward kinematics program

## E.1 Remarks

1. There are 14 types of elements that can be added into a GUI design (eg., Push Button, Slider, Editable Text Box, Static Text Box, etc.).

2. Each element is identified by a unique "Tag" name. The Tag name is referenced in the corresponding MATLAB script for retrieving the value contained in that element, or to set a value to be displayed in that element.

3. Each element can be customized by double-clicking on the element and modifying its properties in the Inspector panel.

# F Appendix VI: Acronyms

| Acronym | Expansion |
|---|---|
| CAD | Computer-Aided Design |
| CAE | Computer-Aided Engineering |
| GUI | Graphical User Interface |
| GUIDE | GUI Development Environment |
| FK | Forward Kinematics |
| IK | Inverse Kinematics |
| XML | eXtensible Markup Language |
| STEP | STandard for Exchange of Product (data) |

# References


[1] Spong, Mark W., Seth. Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. Hoboken, NJ: John Wiley, 2006. Print.

[2] SolidWorks - Wikipedia. *https://en.wikipedia.org/wiki/SolidWorks*. Web, 18 November 2016.

[3] MATLAB - Wikipedia. *https://en.wikipedia.org/wiki/MATLAB*. Web, 18 November 2016.

[4] Simulink - Wikipedia. *https://en.wikipedia.org/wiki/Simulink*. Web, 18 November 2016.

[5] Robotics Toolbox. *http://petercorke.com/Robotics_Toolbox.html*. Web, 19 November 2016.

[6] GUIDE or Programmatic Workflow - MATLAB & Simulink - MathWorks India. *https://in.mathworks.com/help/matlab/guide-or-matlab-functions.html*. Web, 19 November 2016.

[7] Simscape Multibody Link Documentation - MathWorks India. *https://in.mathworks.com/help/physmod/smlink/index.html*. Web, 19 November 2016.

[8] S. Srinivas, G. S. Virk and U. Haider, "Multipurpose supernumerary robotic limbs for industrial and domestic applications," 2015 20th International Conference on Methods and Models in Automation and Robotics (MMAR), Miedzyzdroje, 2015, pp. 289-293