FOUNDATION OF ROBOTICS (ROB-GY6003)
PROJECT 1 REPORT

Rushi Bhavesh Shah (rs7236)

1. Introduction

Serial manipulators are the most common industrial robots which are designed as a series of links connected by motor-actuated joints that extend from a base to an end-effector. Kuka IIWA 14 robot is one such serial manipulator with 7 revolute joints. IIWA stands for "Intelligent Industrial Work Assistant". This project aims to build all the necessary various functionalities for this and similar robot manipulators. This first project aims to build the core fonctions (basic homogeneous transforms, twists, forward kinematics and Jacobians).

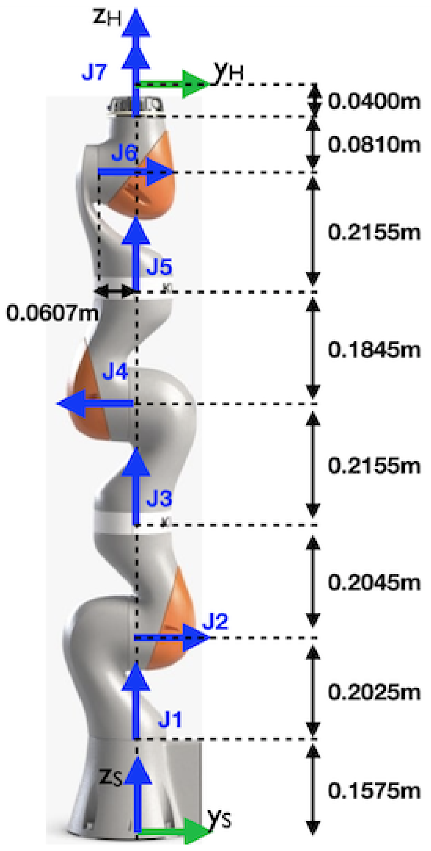

Fig1 : Kuka IIWA 14 robot

In the Fig1, J1, J2, J3, J4, J5, J6 and J7 are all the revolute joints of the robot which rotate about the axis perpendicular to the plane of the joint. This project utilizes robotic python libraries called Pinocchio and Meshcat to display the robot and for other visualization purposes.

2. Methodology

Entire project has been divided into five questions where each question signifies a particular task which needs to be completed in order to achieve final kinematics of the end effector of the robot.

2.1 Basic Functions

This section aimed at writing basic functions required, which will be utilized for the further kinematic calculations of the robot.

(i) *vec_to_skew(w)* - This function transforms a 3D vector (numpy array) into a skew symmetric matrix.

We know, for a vector with three elements $v = [a \;\; b \;\; c\,]$, the skew symmetric* form is

$$v_{\text{skew}} = \begin{bmatrix} 0 & -c & b \\ c & 0 & -a \\ -b & a & 0 \end{bmatrix} \tag{2.1}$$

Hence, we can extract three elements from the vector 'v' and form a new 2D array with the positions of the elements as shown in equation 2.1.

(ii) *twist_to_skew(V)* - This function transforms a 6D twist into a 4x4 matrix.

We know, a 6D twist vector is denoted as

$$V = \begin{bmatrix} w_x \\ w_y \\ w_z \\ v_x \\ v_y \\ v_z \end{bmatrix} \tag{2.2}$$

*A skew symmetric form of a vector is denoted as [v].

which needs to be denoted as a 4x4 matrix in the skew form.

$$V_{ss} = \begin{bmatrix} w_{ss} & v \\ 0 & 1 \end{bmatrix} \tag{2.3}$$

Vss is the skew form of the twist matrix *V,* wherein w_{ss} is the skew symmetric matrix form of the vector formed by the first three elements, which is obtained by using the function *vec_to_skew(w)*. Last three elements of the twist matrix form 'v', which is the velocity vector. To convert the obtained matrix to the homogenous form, an additional row *temp =[0 0 0 1]* is added at the bottom. w_{ss} and *Vss* are concatenated along the column (axis=1) and vector *temp* is added along the (axis = 0). This is done using the *concatenate* function from the numpy library.

(iii) *exp_twist_bracket(V)* - This function returns the exponential of a (bracketed) twist $e^{[V]}$where the input to the function is a 6D twist. This function uses the *expm* function from the scipy library. The function finds out the exponent of a 'square-matrix'. Hence, to make this function work, the 6D vector needs to be converted to the skew symmetric form, and needs to be converted back to the 6D vector after calculating the exponent of the same. For the last step, a new function is defined called *twist_to_skew(V),* which extracts the values w_x , w_y , w_z , v_x , v_y *and* v_z and returns the twist vector as (2.2).
The skew form is calculated using the *twist_to_skew(V)* function. The output is then passed to the *expm* function. Following this, the output is converted back to the twist vector using the function *twist_to_skew(V).*

(iv) *inverseT(T)* - This function returns the inverse of a homogeneous transform T using the *inv* function from the scipy library. For a matrix to be invertible, it should be *square* and *non-singular*.

(v) *getAdjoint(T)* - This function returns the adjoint of a homogeneous transform T. Adjoint of a matrix is *6x6* matrix, given by

$$Adj = \begin{bmatrix} R & 0 \\ [p]R & R \end{bmatrix} \quad T = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix} \tag{2.4}$$

To form the adjoint matrix, R and p are extracted from the transformation matrix. P is converted to the skew symmetric form since further it needs to be multiplied by R. *p* is converted to the skew format using another function called *skew*, since the size of *p* extracted is different and hence can't be passed to the *vec_to_skew(w)* function. Skew of *p* is multiplied with R using the *matmul* function from numpy library. Now, to form the final adjoint matrix, R is first concatenated along columns (axis=1) with a *3x3 zero* matrix and similar concatenation is done for the product of *skew of p* and R. Finally, both the *3x6* matrices are concatenated along rows (axis=0), to obtain a *6x6* Adjoint matrix.

2.2 Forward Kinematics

This part is aimed at writing a function *forward_kinematics(theta)* that gets as an input an array of joint angles and computes the pose of the end-effector.

The pose of the end effector when all the joint angles are *zero* is known. The forward kinematics is calculated using the ***product of exponentials*** method wherein we need the screw matrices for each joint. Since all the joints are revolute joints, angular velocity (*w*) for each joint will be along the axis perpendicular to the joint plane and linear velocity (*v*) will be negative product of *w* and *q* (position of the joint w.r.t the base).

The screw matrices are obtained by stacking the angular and linear velocities of respective joints in a single *6x1* matrix. Respective screw matrices are converted to skew symmetric form using the function twist_to_skew(V) which is multiplied with respective joint angle value. The product is then raised to the power of e. All these individual matrices are multiplied in a sequential manner, starting from the base to the end effector and then multiplied with the original pose of the end-effector. At the end, each element of the output matrix is put in a nested *for loop* for iterating through each row and column, to access each element to round-off the values to four decimal places by using the *round* function.

2.3 Jacobians

This part is aimed at writing a function get_space_jacobian(theta), that computes the space jacobian given an array of joint angles.

The space Jacobian is calculated using the formula given below:

For i = 2, … n

$J_{si}(\theta) = Ad_{e^{[S_1]\theta_{1}} \dots e^{[S_i]\theta_{i-1}}}(S_i)$

(2.4)

with first column $J_{S_1} = S_1$

For the above method, we'll need screw matrices for all the joints, which are already calculated in the previous section. Now, we know from (2.4) that $J_{S_1} = S_1$ and now for J_{S_2}, calculate exponent of product of skew matrix of screw for joint 1 and respective theta, and store in a variable. Then calculate its adjoint (using *getAdjoint(T) function*) and then multiply it with screw matrix of joint 2 i.e. S_2 to obtain the jacobian for the second joint. Similar procedure is followed for subsequent joints, where the adjoint of the product of the exponential matrices (of the product of skew-symmetric matrices and respective theta till the previous joint) is multiplied by the screw matrix of the current joint.

Once all the Jacobian matrices are obtained, all are concatenated in one single matrix along columns (axis=1) using the concatenate function from *numpy* library. At the end, each element of the output matrix is put in a nested *for loop* for iterating through each row and column, to access each element to round-off the values to three decimal places by using the *round* function.

2.4. Displaying hand trajectories

This section aims at computing the position of the hand for all given sets of joint configurations from *joint_trajectory.npy* file, in the spatial frame and display 2D plots for x-y, y-z and x-z positions.

The process starts by declaring three blank lists, each to store the computed x,y and z coordinates of the end effector. A variable is created that has the number of trajectories present in the file. This is done by accessing the second element of the output accessed by using the shape function, which gives the size of data present in the form of matrix in the file.

Furthermore, to calculate the x, y and z coordinates, we need the first three elements of the pose matrix of the end effector, obtained by calculating forward kinematics. These x, y and z values are individually extracted and added to the respective blank lists. This is repeated 200 times by using a *for loop* that iterates for the entire length of the file by using the variable declared earlier. At the end of all 200th iteration, we have three lists with 200 elements each for x, y and z values respectively.

Once these lists are formed, *subplot* function is used from the matplotlib library to create a 3x1 array of plots, to create x-y, x-z and y-z plots. Then all the plots are displayed using the *show* function from the same library.

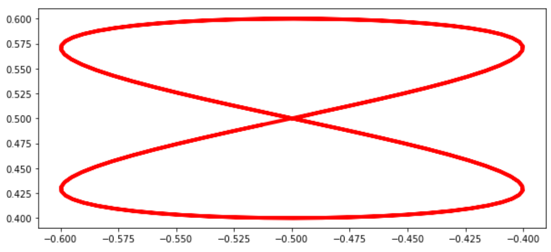

Fig2: X-Y Plot

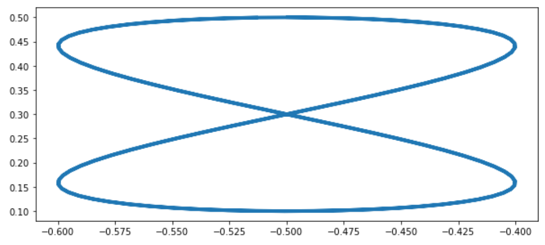

Fig3: X-Z Plot

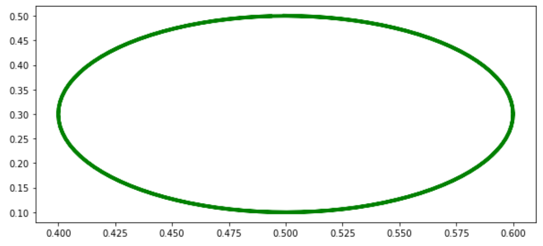

Fig4: Y-Z Plot

2.5 Computing Velocities

This section aims at using Jacobian to compute the linear velocity of the end-effector in: (i) the spatial frame. (ii) the end-effector frame and (iii) in a frame with the same origin as the end-effector frame but oriented like the spatial frame and plotting x, y and z for all the three parts. Also, joint velocities - *dtheta* are extracted from the file joint_velocity.npy, which is the same size as the joint_trajectories.

In the first part, space jacobian is calculated by using the *get_space_jacobian(theta)* function and is multiplied with the set of joint velocities for all the joints, to obtain a ***spatial twist matrix.*** Then the last three elements are extracted from the twist matrix to obtain the linear velocities in x, y and z directions and are added to respective lists declared at the start of the section. This process is repeated for the size of the data in the file. Hence, a *for loop* is iterated 200 times to obtain that many values of x, y and z in their respective lists. Then the graphs are plotted for x, y and z values.

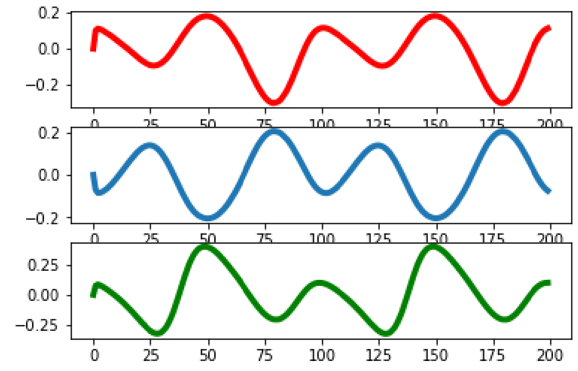

Fig5: Linear velocity of end effector in space frame for X, Y & Z directions

In the second part, in every iteration, Body Jacobian is obtained by multiplying the adjoint of the inverse of the pose of the end-effector obtained from the forward kinematics function. Then it is multiplied with joint velocities to obtain Body Twists. Then, similar to the previous part, linear velocities are extracted and plotted as below.

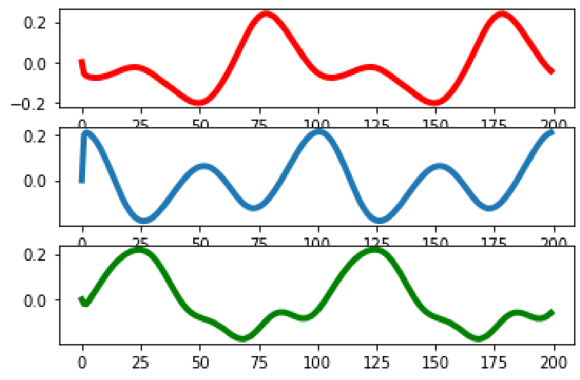

Fig6: Linear velocity of end effector in body frame for X, Y & Z directions

In the third part, since origins of spatial frame and body frame coincide with the same orientations, the R matrix from T (2.4) is changed to *I* (identity matrix). T is the pose of the end effector in the spatial frame obtained by calculating forward kinematics. Then to convert it to the body frame, the matrix is inverted. Now, the space jacobian is calculated and then multiplied with the adjoint of the inverted pose matrix, to find the body jacobian. Then the linear velocity is calculated and plotted as in the previous section. The plot looks like follows.

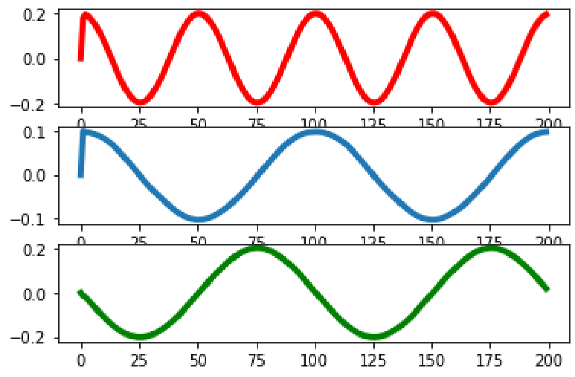
Fig7: Linear velocity of end effector in body frame for X, Y & Z directions

3. Inference

We know, the spatial twist is independent of the selection of the body frame and similarly bady twist is independent of the selection of the spatial frame. From fig. 5 and fig. 6 we can see that both the plots are approximately complementary to each other since the reference frames are inverted. Also, since the selection of frame doesn't affect the twist matrix calculated w.r.t another frame, its non-uniformity can be seen in Fig 5 and Fig 6. However, since the frame of reference has the same orientation as spatial frame and origin as end effector frame, it eliminates the additional linear velocity component caused due to the offset between the origin of the spatial frame and the end effector frame and a smooth sinusoidal plot is obtained. Hence the frame with the same origin as end effector and orientation as spatial frame is more intuitive.

4. Conclusion

Basic functions like *vec_to_skew(w), twist_to_skew(V), exp_twist_bracket(V), inverseT(T), and getAdjoint(T)* were formed to calculate kinematics for Kuka IIWA 14 robot.. These functions are used to calculate the forward_kinematics of the end-effector and the Space Jacobian as well. Furthermore, these are used to display the hand trajectories for a given set of joint angles. Also, the linear velocities of the end effector were calculated and plotted for a set of given joint velocities.