ECE 569

Project Report

Yinglai Wang

Kinematics

Implementation of Forward Kinematics

In my program, forward kinematics is implemented based on PUMA 560 link coordinate transformation matrices.

$$ {}^{i-1}\mathbf{A}_i = \begin{bmatrix} \cos\theta_i & -\cos\alpha_i \sin\theta_i & \sin\alpha_i \sin\theta_i & a_i \cos\theta_i \\ \sin\theta_i & \cos\alpha_i \cos\theta_i & -\sin\alpha_i \cos\theta_i & a_i \sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^{0}\mathbf{T}_i = {}^{0}\mathbf{A}_1 {}^{1}\mathbf{A}_2 \cdots {}^{i-1}\mathbf{A}_i = \prod_{j=1}^{i} {}^{j-1}\mathbf{A}_j \quad ; \quad \text{for } i = 1, 2, \cdots, n $$

joint angle configuration is q = [pi/2 pi/4 0 pi/2 0 0]

DH table

```
Puma 560 (6 axis, RRRRRR, stdDH, fastRNE)
 viscous friction; params of 8/95;
```

j	theta	d	a	alpha	offset
1	q1	0	0	1.571	0
2	q2	0	0.4318	0	0
3	q3	0.15	0.0203	-1.571	0
4	q4	0.4318	0	1.571	0
5	q5	0	0	-1.571	0
6	q6	0	0	0	0

Verify Kinematics

To verify forward kinematics, I compared the result of my implementation (T_formula) and the result that computed by the RST built in function (T_simulate).

To verify inverse kinematics, I used the built in function **ikine6s()** computed the inverse kinematics and forwarded it back using **fkine()**, to check if the result still hold.

Program output:

```
>> kin_verify
verify forward kinematics
T_simulate [double] : 4x4 (128 bytes)
   -1.0000   -0.0000   -0.0000    0.1500
    0.0000   -0.7071   -0.7071    0.0144
    0.0000   -0.7071    0.7071    0.6250
         0         0         0    1.0000

T_formula [double] : 4x4 (128 bytes)
   -1.0000   -0.0000   -0.0000    0.1500
    0.0000   -0.7071   -0.7071    0.0144
    0.0000   -0.7071    0.7071    0.6250
         0         0         0    1.0000

T_simulate = T_formula => forward kinematics verified!
verify inverse kinematics
T [double] : 4x4 (128 bytes)
   -1.0000         0    0.0000    0.1500
   -0.0000   -0.7071   -0.7071    0.0144
    0.0000   -0.7071    0.7071    0.6250
         0         0         0    1.0000

T = T_simulate = T_formula => inverse kinematics verified!
```

Jacobian

Implementation and verification of Jacobian reference to the base coordinate using cross-product method.

$$\begin{bmatrix}\mathbf{v}_n(t)\\ \boldsymbol{\omega}_n(t)\end{bmatrix} \triangleq \begin{bmatrix}\mathbf{v}(t)\\ \boldsymbol{\omega}(t)\end{bmatrix} = \begin{bmatrix}\mathbf{z}_0 \times {}^0\mathbf{p}_n & \mathbf{z}_1 \times {}^1\mathbf{p}_n & \cdots & \mathbf{z}_{n-1} \times {}^{n-1}\mathbf{p}_n \\ \mathbf{z}_0 & \mathbf{z}_1 & \cdots & \mathbf{z}_{n-1}\end{bmatrix} \begin{bmatrix}\dot{q}_1\\ \vdots \\ \dot{q}_n\end{bmatrix}$$

$$= \mathbf{J}(\mathbf{q})\dot{\mathbf{q}}(t) = [\mathbf{J}_1(\mathbf{q}), \mathbf{J}_2(\mathbf{q}), \ldots, \mathbf{J}_n(\mathbf{q})]\,\dot{\mathbf{q}}(t),$$

$$\mathbf{J}_i(\mathbf{q}) = \begin{cases} \begin{bmatrix}\mathbf{z}_{i-1} \times {}^{i-1}\mathbf{p}_n \\ \mathbf{z}_{i-1}\end{bmatrix} & \text{if joint } i \text{ is rotational;} \\ \begin{bmatrix}\mathbf{z}_{i-1} \\ \mathbf{0}\end{bmatrix} & \text{if joint } i \text{ is translational;} \end{cases}$$

joint angle configuration is q = [0 –pi/2 –pi/2 0 pi/6 0]
DH table

```
Puma 560 (6 axis, RRRRRR, stdDH, fastRNE)
 viscous friction; params of 8/95;
```

j	theta	d	a	alpha	offset
1	q1	0	0	1.571	0
2	q2	0	0.4318	0	0
3	q3	0.15	0.0203	-1.571	0
4	q4	0.4318	0	1.571	0
5	q5	0	0	-1.571	0
6	q6	0	0	0	0

In order to use the above equations, I first compute the link transformation matrices ${}^0A_i$ for i = 1,2,3,4,5,6

Then get rotation matrices from corresponding link transformation matrices
${}^0R_i = {}^0A_i(1:3,1:3)$ for i = 1,2,3,4,5,6

Compute the joint axes of motion
$Z_0 = [0\ 0\ 1]^T$ $Z_i = {}^0R_i * Z_0$ for i = 1,2,3,4,5

Get and compute position vector that is needed
${}^0P_i = {}^0A_i(1:3,4)$ for i = 1,2,3,4,5,6
${}^iP_6 = {}^0P_6 - {}^0P_i$ for i = 1,2,3,4,5

Then compute $J_i(q)$ based on the equation
$J_i(q) = [\text{cross}(Z_{i-1}, {}^{i-1}P_6)]$

To verify the result, I compared my result (J_base_formula) and the result that is computed by the RST built in function (T_base_simulate)

Program output:

```
verfy Jacobian reference to base coordinate
J_base_simulate [double] : 6x6 (288 bytes)
    0.1501    0.8636    0.4318         0         0         0
   -0.0203   -0.0000   -0.0000         0         0         0
    0.0000   -0.0203   -0.0203         0         0         0
   -0.0000         0         0    0.0000         0    0.5000
    0.0000   -1.0000   -1.0000   -0.0000   -1.0000   -0.0000
    1.0000    0.0000    0.0000   -1.0000    0.0000   -0.8660

J_base_formula [double] : 6x6 (288 bytes)
    0.1500    0.8636    0.4318   -0.0000         0         0
   -0.0203   -0.0000   -0.0000    0.0000         0         0
         0   -0.0203   -0.0203   -0.0000         0         0
         0         0         0    0.0000         0    0.5000
         0   -1.0000   -1.0000   -0.0000   -1.0000   -0.0000
    1.0000    0.0000    0.0000   -1.0000    0.0000   -0.8660

J_base_simulate = J_base_formula => forward Jacobian reference to base coordinate verified!
```

Implementation and verification of Jacobian reference to the end-effector using differential translation and rotation.

$$\begin{bmatrix} {}^{\mathbf{T}_6}d_x \\ {}^{\mathbf{T}_6}d_y \\ {}^{\mathbf{T}_6}d_z \\ {}^{\mathbf{T}_6}\delta_x \\ {}^{\mathbf{T}_6}\delta_y \\ {}^{\mathbf{T}_6}\delta_z \end{bmatrix}_{dq_i} = \begin{cases} \begin{bmatrix} (\mathbf{p}\times\mathbf{n})_z \\ (\mathbf{p}\times\mathbf{s})_z \\ (\mathbf{p}\times\mathbf{a})_z \\ n_z \\ s_z \\ a_z \end{bmatrix} d\theta_i \\ \begin{bmatrix} n_z \\ s_z \\ a_z \\ 0 \\ 0 \\ 0 \end{bmatrix} dd_i \end{cases} = \begin{cases} \begin{bmatrix} p_x n_y - p_y n_x \\ p_x s_y - p_y s_x \\ p_x a_y - p_y a_x \\ n_z \\ s_z \\ a_z \end{bmatrix} d\theta_i, & \text{if joint } i \text{ is rotational;} \\ \begin{bmatrix} n_z \\ s_z \\ a_z \\ 0 \\ 0 \\ 0 \end{bmatrix} dd_i, & \text{if joint } i \text{ is translational.} \end{cases}$$

$$= \mathbf{J}_i(\mathbf{q}) dq_i.$$

$$\begin{bmatrix} {}^{\mathbf{T}_6}d_x \\ {}^{\mathbf{T}_6}d_y \\ {}^{\mathbf{T}_6}d_z \\ {}^{\mathbf{T}_6}\delta_x \\ {}^{\mathbf{T}_6}\delta_y \\ {}^{\mathbf{T}_6}\delta_z \end{bmatrix} = [\mathbf{J}_1(\mathbf{q}), \mathbf{J}_2(\mathbf{q}), \cdots, \mathbf{J}_6(\mathbf{q})] \begin{bmatrix} dq_1 \\ dq_2 \\ dq_3 \\ dq_4 \\ dq_5 \\ dq_6 \end{bmatrix} = \mathbf{J}(\mathbf{q}) \begin{bmatrix} dq_1 \\ dq_2 \\ dq_3 \\ dq_4 \\ dq_5 \\ dq_6 \end{bmatrix}$$

q and DH table remains the same

Need to compute the U matrices first
$U_6 = {}^5A_6$
$U_{6-i} = {}^{5-i}A_{6-i} * U_{7-i}$ for i = 1,2,3,4,5
Then compute J_i using corresponding U_i based on the follow equation

$$\mathbf{J}_i = \begin{bmatrix} (\mathbf{p} \times \mathbf{n})_z \\ (\mathbf{p} \times \mathbf{s})_z \\ (\mathbf{p} \times \mathbf{a})_z \\ n_z \\ s_z \\ a_z \end{bmatrix} = \begin{bmatrix} p_x n_y - p_y n_x \\ p_x s_y - p_y s_x \\ p_x a_y - p_y a_x \\ n_z \\ s_z \\ a_z \end{bmatrix}$$

To verify the result, I compared my result (J_end_formula) and the result that is computed by the RST built in function (T_end_simulate)

Program output:

```
verify Jacobian reference to end effector
J_end_simulate [double] : 6x6 (288 bytes)
   -0.1299   -0.7377   -0.3638         0         0         0
   -0.0203    0.0000    0.0000         0         0         0
    0.0750    0.4494    0.2335         0         0         0
   -0.5000         0         0    0.5000         0         0
   -0.0000   -1.0000   -1.0000   -0.0000   -1.0000         0
   -0.8660    0.0000    0.0000    0.8660    0.0000    1.0000

J_end_formula [double] : 6x6 (288 bytes)
   -0.1299   -0.7377   -0.3638         0         0         0
   -0.0203    0.0000    0.0000         0         0         0
    0.0750    0.4494    0.2335         0         0         0
   -0.5000         0         0    0.5000         0         0
   -0.0000   -1.0000   -1.0000   -0.0000   -1.0000         0
   -0.8660    0.0000    0.0000    0.8660    0.0000    1.0000

J_end_simulate = J_end_formula => forward Jacobian reference to end-effector verifie
```

For the inverse Jacobian, need to check if the previous result is invertable.
det(J_base_formula) ~= 0
det(J_end_formula) ~= 0
Both invertable.
Then compute **inv_J_base = inv(J_base_formula)**
inv_J_end = inv(J_end_formula)

Verify inverse Jacobian by using following properties.

1) $\mathbf{J}(\mathbf{q})\mathbf{J}^{+}(\mathbf{q})\mathbf{J}(\mathbf{q}) = \mathbf{J}(\mathbf{q})$

2) $\mathbf{J}^{+}(\mathbf{q})\mathbf{J}(\mathbf{q})\mathbf{J}^{+}(\mathbf{q}) = \mathbf{J}^{+}(\mathbf{q})$

3) $[\mathbf{J}(\mathbf{q})\mathbf{J}^{+}(\mathbf{q})]^{T} = \mathbf{J}(\mathbf{q})\mathbf{J}^{+}(\mathbf{q})$

4) $[\mathbf{J}^{+}(\mathbf{q})\mathbf{J}(\mathbf{q})]^{T} = \mathbf{J}^{+}(\mathbf{q})\mathbf{J}(\mathbf{q})$

I picked the first property to check the result
J_base = J_base_formula*inv_J_base*J_base_formula;
J_end = J_end_formula*inv_J_end*J_end_formula;

Program output:

```
Matrices are not singular, invertable
J_base [double] : 6x6 (288 bytes)
    0.1500    0.8636    0.4318    0.0000   -0.0000   -0.0000
   -0.0203   -0.0000   -0.0000    0.0000   -0.0000   -0.0000
   -0.0000   -0.0203   -0.0203    0.0000   -0.0000    0.0000
   -0.0000   -0.0000   -0.0000    0.0000    0.0000    0.5000
   -0.0000   -1.0000   -1.0000   -0.0000   -1.0000   -0.0000
    1.0000    0.0000    0.0000   -1.0000    0.0000   -0.8660

J_end [double] : 6x6 (288 bytes)
   -0.1299   -0.7377   -0.3638   -0.0000   -0.0000   -0.0000
   -0.0203    0.0000    0.0000   -0.0000   -0.0000    0.0000
    0.0750    0.4494    0.2335    0.0000    0.0000    0.0000
   -0.5000   -0.0000   -0.0000    0.5000   -0.0000    0.0000
   -0.0000   -1.0000   -1.0000   -0.0000   -1.0000   -0.0000
   -0.8660    0.0000    0.0000    0.8660    0.0000    1.0000

J_base = J_base_formula
J_end = J_end_formula
result satisfies the formula J=J*(J+)*J
```

Draw circle using Kinematics and Jacobian

Implementation of draw circle by Kinematics

In order to use kinematics to drive PUMA 560 draw a circle, need to get the trajectory first.
The function **my_circle(center,r)** will take **center** and **radius** as its parameter and return the **trajectory**.

Then use RST built in function **transl(trajectory^T)** to translate trajectory to 4x4 homogeneous matrices.

Using inverse kinematics to get solution **q**
q = p560.ikine6s(transl(trajectory^T))

Generate graphical result.
p560.plot(q,'fps',50/(0.5*pi/0.4),'trail','b')

Using Kinematics is just make the robot point to all points falls on the trajectory, it is hard to ensure the end-effector moves at the desired speed.
In my program, I computed the total length of path (2*pi*r), then divided by the desired speed 0.4m/s, this would be the total time that is needed to complete the circle.
I used 50 points on the trajectory, therefore there will be 50 total time frames. By control the **fps** rate to (50/(2*pi*r/0.4)), the desire speed of 0.4m/s is ensured.

Implementation of draw circle by Jacobian

Draw this circle using Jacobian need to know the velocity vector first.
The center of the circle is **(0.5,-0.25,0)** and radius is **0.25**, I picked **(0.5,-0.25,0)** as the start point.

In my program, function **draw_circle_jac(start_point,r,speed)** will take start point, radius, speed as its parameter and return the solution **q**.

The velocity is calculated as follows: (based on **theta=l/r**)
v=[0;velocity*sin(velocity*t/r);velocity*cos(velocity*t/r);0;0;0]
My time interval is **deltat=0.02**
Based on the time interval, total number of **q** is computed
round(2*pi*r/velocity/0.02)+1
Then the first **q** can be compute based on the start point
q**(1,:)=p560.ikine6s(transl(start_point))**
In order to compute the future q, need to get q' based on this equation

$$\dot{\mathbf{q}}(t) \equiv \mathbf{J}^{+}(\mathbf{q})\dot{\mathbf{x}}(t)$$

q'=inv(p560.jacobn(q(1,:)))*v

Then the following procedures will loop **round(2*pi*r/velocity/deltat)** times

q(i+1)=q(i)+q'*deltat
updating q' based on q(i)
increment time
updating v based on time

In this way, solution q can be computed.
p560.plot(q,'fps',1/0.02,'trail','b') will generate the graphical result as before.

In this Jacobian approach, it is easy to ensure it moves at a given speed, because the solution q is computed based on the velocity we calculated, and the velocity is computed based on given speed 0.4m/s.
Using Jacobian makes the graphical result looks smoother than use kinematics since it is actually moving. But it might lack of accuracy compare to using kinematics, to make it more accurate, I used deltat=0.02, so there would be total number of approximately 200 time frames.
Fps is also controlled to ensure a better visualization.

Implementation of draw line by Kinematics

This program will take 2 user input: **a** and **b**
I picked a=1, b=0.5
Start and end point is the intersection with x-axis and y-axis
Then drive the robot to draw a line along **y=ax+b** with **z=-0.4** with a constant speed 0.1m/s

In order to use kinematics, need to get the path first.
I used the built in function **ctraj(start,end,50)** to get a Cartesian path.

Use inkine6s() to get solution q
To ensure a better performance, I eliminated all points that can not be reached by checking if q is NaN.

Generate graphical result
p560.plot(new_q,'fps',(# of points)*0.1/norm(start-dest),'trail','b')
Fps is also controlled to ensure a better visualization.

Speed is ensured using the same way as using kinematics draw circle.

Implementation of draw line by Jacobian

Parameter used in this program is as same as used in draw line by kinematics.
deltat is 0.02 same as using Jacobian draw circle

First, calculate velocity based on the given speed
v=[velocity*(1/sqrt(1+a*a));velocity*(a/sqrt(1+a*a));0;0;0;0]

Based on the time interval, total number of **q** is computed

round(2*pi*r/velocity/0.02)+1
Then the first **q** can be compute based on the start point
q**(1,:)=p560.ikine6s(transl(start_point))**
In order to compute the future q, need to get q'
q'=inv(p560.jacobn(q(1,:)))*v

Then the following procedures will loop **round(norm(start-end)/velocity/deltat)** times
q(i+1)=q(i)+q'*deltat
updating q' based on q(i)
increment time
(unlike drawing circle, v doest not change since it is a straight line)

In this way, solution q can be computed.
p560.plot(q,'fps',1/0.02,'trail','b') will generate the graphical result as before.

Robot arm moves at the given speed is ensured since we used Jacobian and computed velocity based on given speed 0.1m/s. Because the total number of time frame is large, fps is controlled to make a better visualization.

Conclusion
Overall, the Jacobian approach for both drawing are smoother than using kinematics, but need a large number of time frames to improve its accuracy.
Using kinematics is easier to implement and more accurate than using Jacobian, but less smoother. This can also be improved by adding more points on the trajectory.
The graphical result might slow down due to the large amount of calculation or time frame. Fps control is needed to have a better visualization.