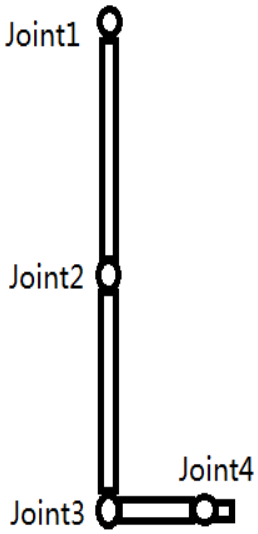
Design of Robotic Systems

Forward and Inverse Kinematics Simulation

Lin Li

1. Introduction

In this lab, I chose a human leg as my real-world linkage and tried to use MATLAB to simulate its forward and inverse kinematic movements.



I used 4 revolute joints to simulate this specific body part: joint1 represents the hip joint, joint2 represents the knee joint, joint3 represents the ankle joint and joint4 represents the toe joint.

And I defined the link between joint1 and joint2 as ***Leg_Link_1*** which is the thigh, the link between joint2 and joint3 as ***Leg_Link_2*** which is the shin, the link between joint3 and joint4 as ***Leg_Link_3*** which is the sole, and the final link as ***Leg_Link_4*** which is the toe.

In this specific case, since we are assuming each of the joint is a 1-DOF joint in this experiment, the end effector of this robotic leg will always be moving within a single plane, which is exactly like the movement of a leg when a human is walking straight forward.

In other words, we can assume that it will stay in the Z=0 plane. Therefore, only X and Y coordinates of the end effector will stay as variables when Z is a constant 0 during the simulation. And we also don't need to consider other rotational states of the end effector.

The X, Y coordinates of the end effector will be completely determined by the states of the 4 joint angles and their limitations.

For example, in the initial states, I defined the 4 angles in the code as 0,0,90,0. With these states, the end effector will be pointing at (X,Y)=(38,14).

Now, if I want to move the end effector to another point , say for example (X,Y)=(-28,36). I will need to figure out another specific combination values for the 4 joints so that it would move the end effector from (X,Y)=(38,14) to (X,Y)=(-28,36), which is, by definition, the inverse kinematic movement.

The process and result of this experiment will be shown later in the result section of the report.

2. Methods

I measured an adult male's leg(which is mine) to get the natural ratio of the lengths of the links. The length from link1 to link4 were approximately 20 inches, 18 inches, 9 inches and 2 inches.

So I defined the corresponding constants in the MATLAB codes accordingly as below.

```
139 -    Leg_Link_1=20;
140 -    Leg_Link_2=18;
141 -    Leg_Link_3=9;
142 -    Leg_Link_4=5;
143
```

After testing the code several times, I decided to use 5 instead of 2 for the value of Leg_Link_4 to get a more significant simulation graph.

Considering each joint of a human leg has its own rotational limitation, I also roughly measured the range of the reachable angles of each joint of my leg and recorded them as below.

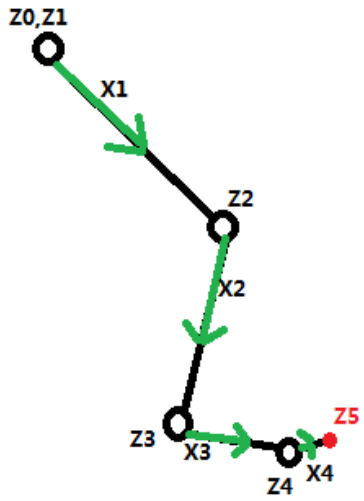
Joint num	Joint1	Joint2	Joint3	Joint4
Angle range	-30° to 135°	-150° to 0°	45° to 110°	-15° to 60°

And I implemented these angle limits as a function called "angle_limit" in the MATLAB as below to make the boundaries of the simulation.

```
106
107     function output = angle_limit( an1,an2,an3,an4 )
108 -   if an1>=-pi/6&&an1<=3*pi/4&&an2>=-5*pi/6&&an2<=0&&an3>=pi/4&&an3<=11*pi/18&&an4>=-pi/12&&an4<=pi/3
109 -       output=1;
110 -   else
111 -       output=0;
112 -   end
```

Basically, what this function does is that, it will return 1 if all input angles are within the natural range of human leg joints, otherwise it will return 0.

Apparently, with the link length constants, this function plays a very important role in the code to make the simulation object look like a real human leg.



I determined the D-H parameters according to this schematic of the leg.

Joint_i	a_i-1	α_i-1	d_i	θ_i
1	0	0	0	Angle_1
2	Leg_Link_1	0	0	Angle_2
3	Leg_Link_2	0	0	Angle_3
4	Leg_Link_3	0	0	Angle_4
5(end effector)	Leg_Link_4	0	0	0

In the D-H parameter table, Leg_Link_i is corresponding to the values I explained previously, and the Angle_i is the angle variable that can be determined by the user in the simulation.

With the D-H parameters, I implemented the forward kinematic function in the MATLAB as below.

```
115    function T_05 = forward_kinematic( an1,an2,an3,an4 )
116 -    T_01=[cos(an1) -sin(an1) 0 0;sin(an1) cos(an1) 0 0;0 0 1 0;0 0 0 1];
117 -    T_12=[cos(an2) -sin(an2) 0 20;sin(an2) cos(an2) 0 0;0 0 1 0;0 0 0 1];
118 -    T_23=[cos(an3) -sin(an3) 0 18;sin(an3) cos(an3) 0 0;0 0 1 0;0 0 0 1];
119 -    T_34=[cos(an4) -sin(an4) 0 9;sin(an4) cos(an4) 0 0;0 0 1 0;0 0 0 1];
120 -    T_45=[1 0 0 5;0 1 0 0;0 0 1 0;0 0 0 1];
121 -    T_05=T_01*T_12*T_23*T_34*T_45;
122
```

What this function does is that it will take the joint states, which are 4 angles, as its arguments and return the desired transformation matrix T_05 which consists of the X,Y coordinates of the end effector in its 4^{th} column that is corresponding to this specific angle state.

If we let the angles stay as variables in the matrix and do the multiplication, we can express X and Y as functions in terms of the angles as below.

$$X=L_1Cos(\theta_1)+L_2Cos(\theta_1+\theta_2)+ L_3Cos(\theta_1+\theta_2+\theta_3)+L_4Cos(\theta_1+\theta_2+\theta_3+\theta_4)$$

$$Y= L_1Sin(\theta_1)+L_2Sin(\theta_1+\theta_2)+ L_3Sin(\theta_1+\theta_2+\theta_3)+L_4Sin(\theta_1+\theta_2+\theta_3+\theta_4)$$

Since the sample space of the angle in this case is not too large, I decided to take the advantage of it in the implementation of the inverse kinematic function.

```
123
124    function Angles = inverse_kinematic(X,Y)
125
```

……

……

……

```
170 -       sum2=tem_a1(i)+tem_a2(j);
171 -       sum3=tem_a1(i)+tem_a2(j)+tem_a3(k);
172 -       sum4=tem_a1(i)+tem_a2(j)+tem_a3(k)+tem_a4(l);
173 -       test_X=floor(5*cos(sum4)+9*cos(sum3)+18*cos(sum2)+20*cos(tem_a1(i)));
174 -       test_Y=floor(5*sin(sum4)+9*sin(sum3)+18*sin(sum2)+20*sin(tem_a1(i)));
175
176 -       if test_X==X&&test_Y==Y
177 -       Angles(1)=tem_a1(i);
178 -       Angles(2)=tem_a2(j);
179 -       Angles(3)=tem_a3(k);
180 -       Angles(4)=tem_a4(l);
181
```

As we can see, the inverse kinematic function will take X and Y coordinate as its arguments and return a vector “Angles” that consists of the corresponding angles for the given coordinate.

Basically, what this function does is that, it will divide the range of the angles and scan each possible combination starting from the median value of each range until the result matches the given coordinates or it runs out of the range.

Keep checking the command window of the MATLAB after clicking the inverse kinematic button on the UI. If there is a solution, it will display “Success!” on the command window (not on the UI window), otherwise it will display “No solution!”.

If nothing’s been displayed, that means the program is still running.

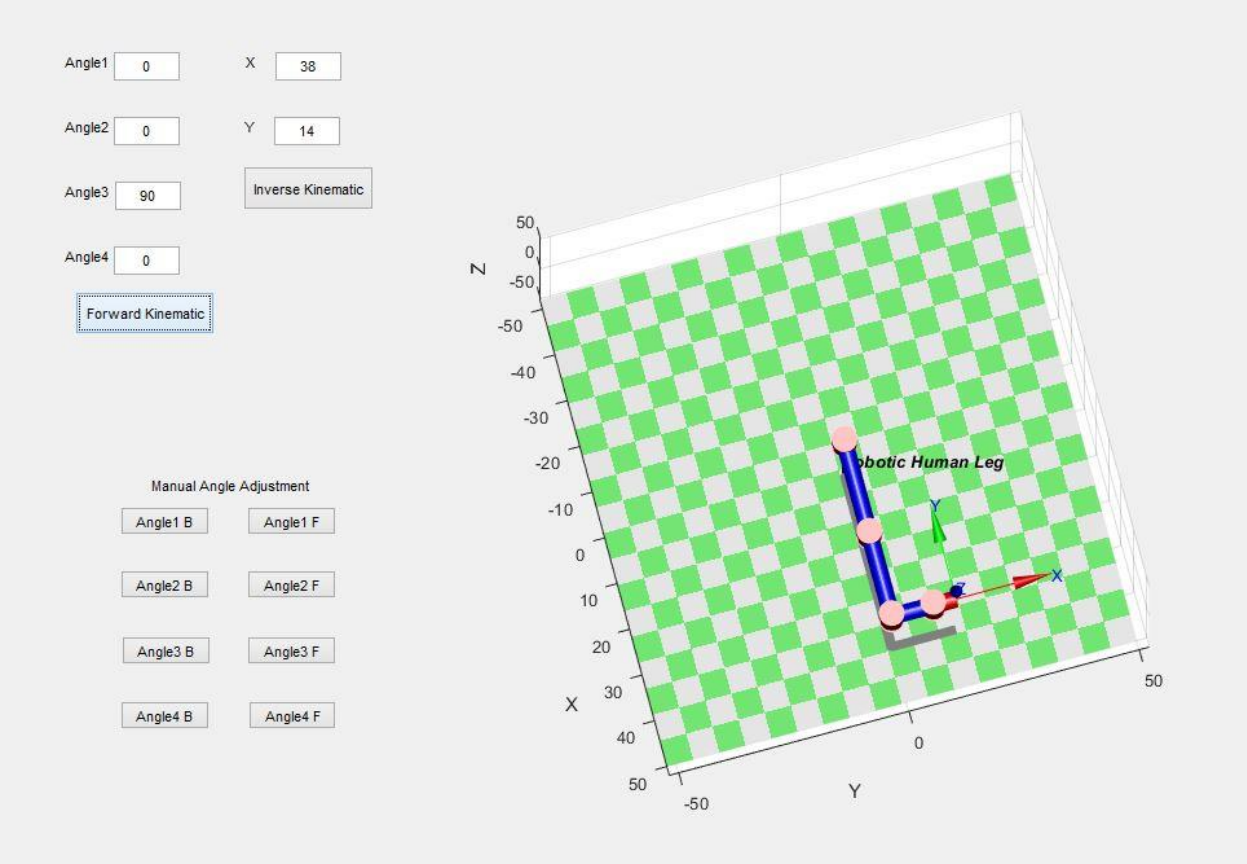
It might take some time if the coordinate is “bad”, but it won’t take too long.

To see the full implementation details of the MATLAB code, please check my Github.

https://github.com/Space-Tortoise/UCLA-EE183DA-2018-Winter/tree/master/Lab1

3. Results

The MATLAB code will initialize the simulation UI as below.

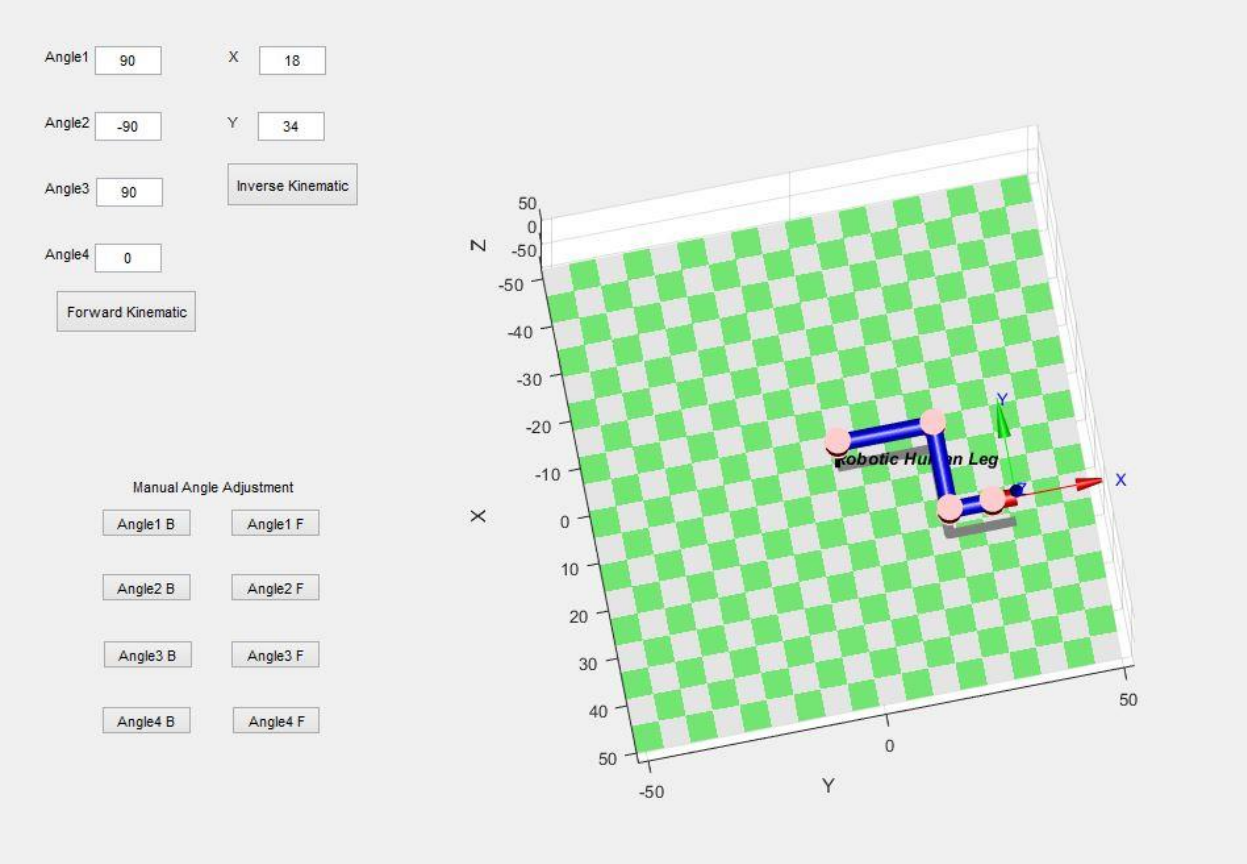


As I described previously in the intro section of the report, the angles of the joints will be initialized as [0 0 90 0], and the end effector will be pointing at (X,Y)=(38,14) as it can be seen in the picture.

I chose this state to be the initial state because it looks like the most natural state of a human leg.

If I plug in a set of angles and click “Forward Kinematic” button, the joints will rotate to the desired angle and it will also display the coordinate of the end effector.

For example, if I plug in [90 -90 90 0] and click “Forward Kinematic” button, the result will look like this.

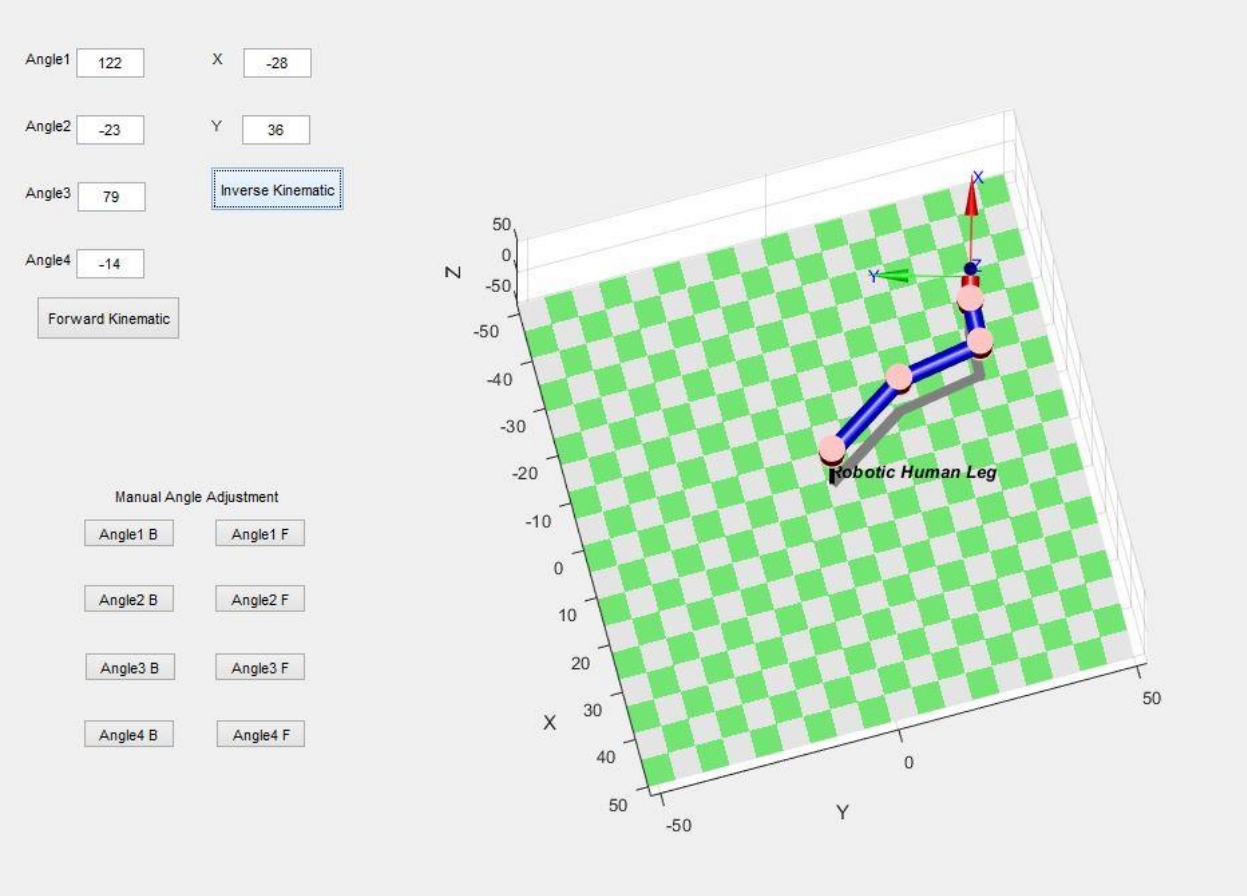


So in this case, the state coordinate variables of the end effector is calculated by the code to be (X,Y)=(18,34).

“Forward Kinematic” is also filtering out the input angles that are out of the defined range as I explained previously. If the angle are illegal, it won’t simulate.

Let’s go back to the initial state [0 0 90 0] and test the case we talked about earlier in the intro section. Suppose I want to move the end effector from (X,Y)=(38,14) to (X,Y)=(-28,36).

In this case I can plug -28 and 36 into the X and Y box on the UI and click “Inverse Kinematic” button. Then the result will be as below.



It returns the corresponding angle state [122 -23 79 -14] and moves the end effector to the desired position by adjusting each angle to the values of the angle state.

I also added some manual angle controlling buttons just for fun. With these buttons we can easily control the joints in any way we like.

If you actually run the code and watch the dynamic simulation, you will see how the trajectory works. The way I did it is that I used a function to divide the difference between the starting angle and destination angle into 20 even pieces.

For example, if angle1 is changing from 0 to 60 and angle2 is changing from 0 to -80, it will make each move as 3 degree for angle1 and -4 degree for angle2. Then it will plot using forward kinematic method for the angle state of [0 0 . .] , [3 -4 . .] , [6 -8 . .] , [9 -12 . .] …… [57 -76 . .] , [60 -80 . .] in 20 steps during a certain period of time.

Apparently, by this method, the end effector would most likely move along a curvy path instead of a straight line.

Comparing with the straight line path, I didn't see any advantage or disadvantage of taking a curvy path. Because in this specific case, the straight line path won't make the movement necessarily quicker, easier or more efficient.

It make sense because in the real world, the toe is not intentionally moving along a straight line when we are walking, either.

Therefore, unless there is a space limitation, for example the space is blocked in some way so that the only way the end effector move is along a straight line, the trajectory defined by this code is pretty efficient and accurate.