Sapienza University of Rome

Space Robotics

Department of Space and Astronautical Engineering

3D Inverse Kinematics realtime solver for n-Link Robotic Manipulator

Author:
Siddharth Deore

Tutor:
Prof. Fabio Santoni

January 23, 2019

Index

Nomenclature

Acronyms

CCD	Cyclic Coordinate Descent
DLS	Damped Least Square
FABRIK	Forward and Backward Reaching Inverse Kinematics
FTL	Follow the leader
IK	Inverse Kinematics
SVD-DLS	Pseudo Inverse Damped Least Square

Symbols

θ_i	Revolute joint variable of index i
d_i	Prismatic joint variable of index i
O_i	Position vector in euclidean reference system
q_i	Joint variable of index i
R	Rotation Matrix
T	Homogeneous Transformation Matrix

Chapter 1

Introduction

Due to considerable increase in automation since last couple of decades, robots became essential part of our life. We are surrounded by automated system ranging from Industrial assembly line robot to spacecraft docking arm. Since tasks that robot can perform are getting more complicated and it is difficult to find analytical solution for such closely coupled nonlinear systems. Even simplest 2D robotic configuration having only two degree of freedom (two links l_1,l_2 and two joint variable θ_1,θ_2) has two analytical solutions [5] in closed form complexity further increases with increased degree of freedom. With availability of faster computing units, it is possible to compute solution with numerical iterative procedures in real-time, making system more responsive to surrounding environment. Goal of project is to compute Forward Kinematics and Inverse Kinematics of 3 Dimensional robotic manipulator whose structure and configuration might change in real time, a universal iterative solver which can be used with any robotic arm with minimum possible efforts. The term Forward Kinematics is Computing the position of the end effector from known joint positions, orientation and link lengths. Contrarily Inverse Kinematics is estimation of the joint states necessary to reach desired position. Inverse kinematics has great deal of applications such as moving arm of Da Vinci's Robotic knight (Fig 1.1) or to find the joint angle in SpotMini's (Fig 1.2) legs to balance itself or to walk around.

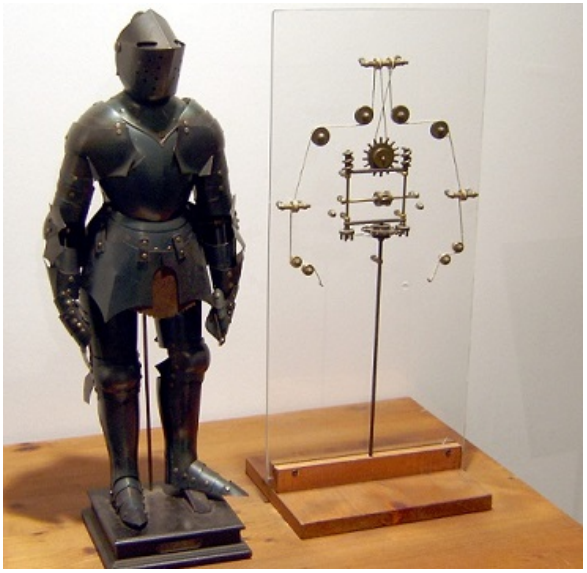

Figure 1.1: L'Automa Cavaliere di Leonardo da Vinci

Figure 1.2: Boston Dynamics Spot Mini

1.1 A note on Forward Kinematics

A robotic manipulator is composed of links connected with joints, a joint can be made simple one such as prismatic or pivot, or can be complex form of both type such as spherical or ball socket joint. We can model any type of complex joint with set of revolute and prismatic assuming revolute joint can be described by angle of rotation and prismatic joint by it's displacement (Fig 1.3).

Now manipulator with n joints will have $n+1$ links, we will denote joint variable with q_i where $i = 1$ to n and link index will be $i = 0$ to $i-1$ where link $i = 0$ is base or anchored to fix reference frame and link $i = n-1$ is End-Effector. Co-ordinate reference frame $o_i x_i y_i z_i$ is attached to link i, frame $o_0 x_0 y_0 z_0$ is attached to base link is inertial reference frame. Fig 1.4 shows reference frame attached joints of elbow manipulator.

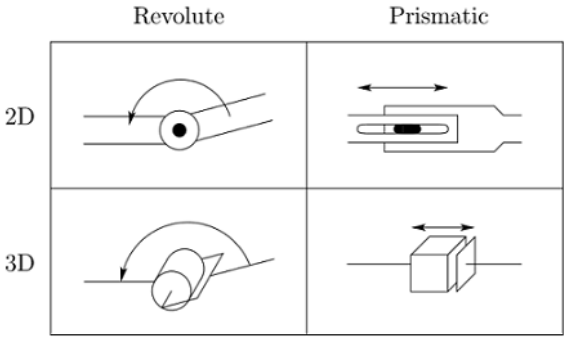

Figure 1.3: Symbolic Representation of Robotic Joints

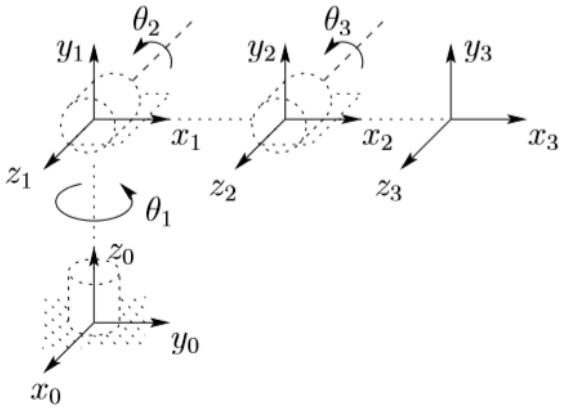

Figure 1.4: Coordinate frames attached to elbow manipulator.

Homogeneous transformation matrix T_i expresses position and orientation of $o_i x_i y_i z_i$ with respect to $o_{i-1} x_{i-1} y_{i-1} z_{i-1}$, T_i is function of only joint variable q_i

$$T_i = T_i(q_i) \tag{1.1}$$

now homogeneous transformation matrix that express position and orientation of $o_j x_j y_j z_j$ with respect to $o_i x_i y_i z_i$ is called as rototranslation matrix T_j^i if $i < j$

$$T_j^i = T_{i+1} T_{i+2} T_{i+3} ... T_{j-1} T_j \qquad \iff i < j \tag{1.2}$$

$$T_j^i = I \qquad \iff i = j \tag{1.3}$$

$$T_j^i = (T_j^i)^{-1} = T_i^j \qquad \iff i > j \tag{1.4}$$

Position and orientation of end effector with respect to base can be realized by Homogeneous transformation matrix

$$T = \begin{bmatrix} R_n^0 & O_n^0 \\ 0 & 1 \end{bmatrix} \qquad T_j^i = \begin{bmatrix} R_j^i & O_j^i \\ 0 & 1 \end{bmatrix} \tag{1.5}$$

The rotation matrix R_i^j is obtained from rotating z-axis by angle α followed by rotating y-axis by angle β and finally rotating x-axis by angle γ. We can select any rotation sequence considering correct rotation matrix but have to consistently and carefully maintain it throughout computation.

$$R_z(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad R_y(\beta) = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \quad R_x(\gamma) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix} \tag{1.6}$$

A complete rotation can be obtained directly from Equation 1.7, which is also identified as Euler 3-2-1 rotation sequence.

$$R(\alpha,\beta,\gamma) = R_z(\alpha)\, R_y(\beta)\, R_x(\gamma) = \begin{bmatrix} \cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma \\ \sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma \\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma \end{bmatrix}. \tag{1.7}$$

Now we can obtain Homogeneous transformation matrix of 3D bodies from rotation $R(\alpha, \beta, \gamma)$ followed by translation x_t, y_t, z_t

$$T = \begin{bmatrix} \cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma & x_t \\ \sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma & y_t \\ -\sin\beta & \cos\beta\sin\gamma & \cos\beta\cos\gamma & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.8}$$

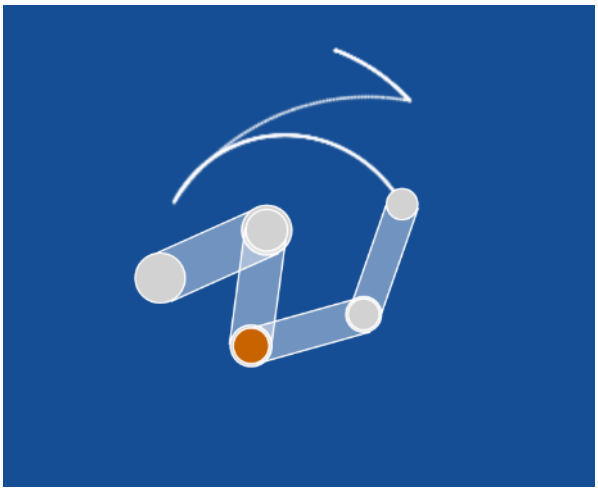

Figure 1.5: Path Traced by planner robot with 4 links using forward kinematics

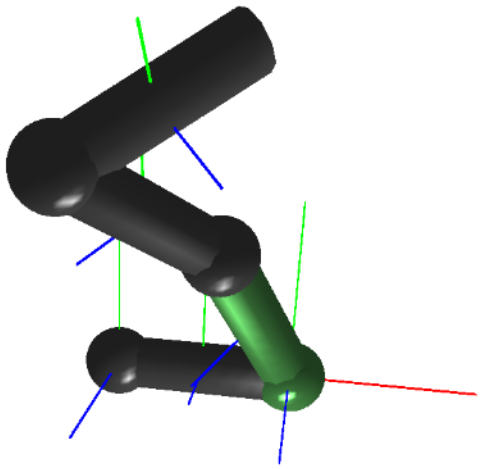

Figure 1.6: 3D Forward kinematics post processing with WEBGL and JavaScript

1.2 Denavit Hartenberg Representation

Denvit Harteberg or D-H representation is commonly used convention for selecting frame of reference in such way that each homogeneous transformation T_i is product of four basic transformations.

$$T_i = \begin{bmatrix} cos\theta_i & -sin\theta_i & 0 & 0 \\ -sin\theta_i & cos\theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\alpha_i & -sin\alpha_i & 0 \\ 0 & -sin\alpha_i & cos\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.9}$$

$$T_i = \begin{bmatrix} cos\theta_i & -sin\theta_i cos\alpha_i & sin\theta_i sin\alpha_i & a_i cos\theta_i \\ sin\theta_i & cos\theta_i cos\alpha_i & -cos\theta_i sin\alpha_i & a_i sin\theta_i \\ 0 & sin\alpha_i & cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.10}$$

In Equation 1.10 the four parameters a_i, α_i, d_i, and θ_i are known as link length, link twist, link offset, and joint angle, respectively. These joint parameters are associated with i^{th} link and i^{th} joint. The function `getTransferforMatrix()` in Code 1.2 takes DH parameters as arguments and returns transformation matrix.

```
function DHParameter(theta,d,a,alfa) {
    this.theta=theta;    // Joint Angle
    this.d=d;            // Link Offset
    this.a=a;            // Link Length
    this.alfa=alfa;      // Link Twist
}
```

Code 1.1: JavaScripts ES6 prototype of DH parameters

```
function getTransformMatrix(theta, d, a, alpha){
    let r11 = cos(theta);
    let r12 =-sin(theta)*cos(alpha);
    let r13 = sin(theta)*sin(alpha);
    let r21 = sin(theta);
    let r22 = cos(theta)*cos(alpha);
    let r23 =-cos(theta)*sin(alpha);
    let r31 = 0;
    let r32 = sin(alpha);
    let r33 = cos(alpha);
    let dx  = a*cos(theta);
    let dy  = a * sin(theta);
    let dz  = d;
    /*   _                  _
         | r11 r12 r13 dx  |
    T=   | r21 r22 r23 dy  |
         | r31 r32 r33 dz  |
         |_  0   0   0   1_|
    */
    let T=math.matrix([[r11,r12,r13,dx],[r21,r22,r23,dy],[r31,r32,r33,dz
        ],[0,0,0,1]]);

    return T;
}
```

Code 1.2: Function to compute transformation matrix from DH parameters

Chapter 2

Literature Survey

Most commonly discussed numerical solution to Inverse kinematics is based on Jacobian of system.

2.1 Jacobian Transpose

Jacobian is basically a matrix of partial derivatives of entire system which defines how end effector changes relative to instantaneous changes in joint variables. Consider end effector vector $\mathbf{e} = [x\, y\, z]^T$ and joint variable vector $\mathbf{q} = [\theta_1\, \theta_2 \ldots \theta_n]^T$ then Jacobian of system is given by:

$$J = \left[\frac{\partial \mathbf{e}}{\partial \mathbf{q}}\right] = \begin{bmatrix} \frac{\partial x}{\partial \theta_1} & \frac{\partial x}{\partial \theta_2} & \cdots & \frac{\partial x}{\partial \theta_n} \\ \frac{\partial y}{\partial \theta_1} & \frac{\partial y}{\partial \theta_2} & \cdots & \frac{\partial y}{\partial \theta_n} \\ \frac{\partial z}{\partial \theta_1} & \frac{\partial z}{\partial \theta_2} & \cdots & \frac{\partial z}{\partial \theta_n} \end{bmatrix} \tag{2.1}$$

$$\dot{\mathbf{e}} = J\dot{\mathbf{q}} \;\rightarrow\; \dot{\mathbf{q}} = J^{-1}\dot{\mathbf{e}} \tag{2.2}$$

2.2 Pseudo Inverse

Problem jacobian is we can not grantee if it is invertible so generally pseudo inverse is computed for non square matrix.

$$J^{+} = (J^T J)^{-1} J^T \tag{2.3}$$

since jacobian is linear approximation of nonlinear system we have to iterate taking small steps. Iterative procedure to compute inverse kinematics solution is discussed by Meredith [6]. Furthermore The damped least squares method also known as Levenberg-Marquardt method avoids many of the pseudoinverse method's problems with singularities was first used for inverse kinematics by Wampler [9] and Nakamura and Hanafusa. It is related to change in joint variable. In our case joint variable $\mathbf{q}$ is composed of θ_i, and change in θ is computed as:

$$\Delta\theta = J^T(JJ^T + \lambda^2 I)^{-1}\mathbf{e} \tag{2.4}$$

2.3 Single Value Decomposition

The singular value decomposition (SVD) is method used to analyze the pseudoinverse and the damped least squares methods discussed by Buss [3].

2.4 Cyclic Coordinate Descent (CCD)

CCD algorithm was first propesed by Wang and Chen [10], in this method outermost joints are moved first in order to reach target, difference in joint position p_c - endeffector position p_e and joint position - target position p_t is calculated and then rotations to reduce this difference to zero. For every joint p_c is rotated by θ about axis $\bar{r}$.

$$cos(\theta) = \frac{p_e - p_c}{||p_e - p_c||} \cdot \frac{p_t - p_c}{||p_t - p_c||} \tag{2.5}$$

$$\bar{r} = \frac{p_e - p_c}{||p_e - p_c||} \times \frac{p_t - p_c}{||p_t - p_c||} \tag{2.6}$$

$$\theta = \frac{p_e - p_c}{||p_e - p_c||} \cdot \frac{p_t - p_c}{||p_t - p_c||} \tag{2.7}$$

2.5 Triangulation

Muller [7] proposed this algorithm which iterates through every joint to the end-effector first by rotating most significant joint like human arm and has good application in character animation.

2.6 Follow The Leader (FTL)

Brown et. al. [2] used this algorithm to simulate knot tying rope, In this algorithm base is not fixed and entire kinematic chain consisting N links is free floating. Leader link is dragged towards goal this is done by rotating $Link_i$ from its joint towards goal and then moving link towards goal, all other links follows next links.

2.7 FABRIK

Forward and Backward Reaching Inverse Kinematics Algorithm is introduced by Andreas Aristidou and John Lasenby [1] and compared it with various above mentioned algorithms. FABRIK starts from last joint of chain and works forward adjusting each joint as in free chain is dragged towards goal by using FTL method then same chain is dragged backward towards fixed base using FTL both Forward and Backward reaching is iterated one after another till error between goal and end effecter is minimized.

For single kinematic chain with 10 joints FABRIK takes least execution time to converge solution among above mentioned methods followed by FTL and Triangulation. On the other hand Jacobian Transpose takes most time to converge for both reachable and unreachable targets. [1]

Chapter 3

Approach

In order to implement numerical scheme for inverse kinematics we need to define data structure of elements. A two dimensional kinematic chain is composition of N links $link_i$ of length l_i, connected at rotating or pivot joints which can freely rotate by angle θ_i. Each link has two pivot points namely $A_i(x, y)$ and $B_i(x, y)$ separated by length l_i and chain is configured such a way that each next links $A_{i+1}(x, y)$ is connected to $B_i(x, y)$ shown in Fig. 3.1.

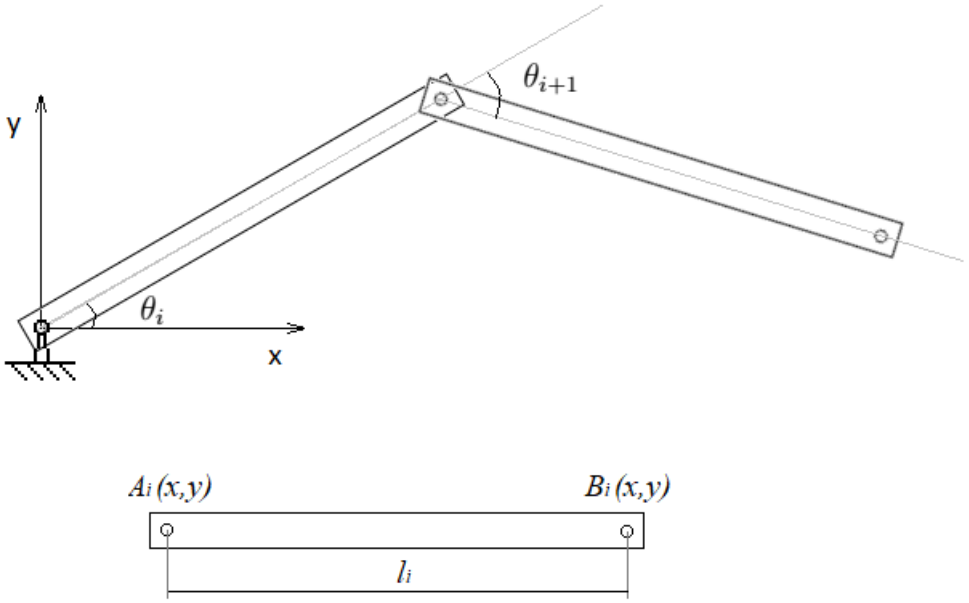

Figure 3.1: 2D kinematic chain configration

Let us consider a chain is not connected to any fixed base and free to move. Starting with last $link_N$, to reach desired target coordinate $T(x, y)$ first compute heading angle between $A_N(x, y)$ and $T(x, y)$ and rotate link in such a way that

$$heading = tan^{-1}\left(\frac{dy}{dx}\right) \tag{3.1}$$

here difference point $A_N(x, y)$ and $T(x, y)$ in x and y direction is dx and dy. Now move link towards target without chaining it's *heading* in such a way that new coordinate of $A_N(x, y)$ equals $T(x, y)$. Now for each subsequent link $N - 1$ move to new target is $A_N(x, y)$ till first link of chain. Figure 3.2 shows 4 link chain dragged forward towards goal and then dragged backwards towards towards base. Entire kinematic chain needs to iterate both forward and backward reaching scheme till desired error tolerance to reach goal is appreciable for application.

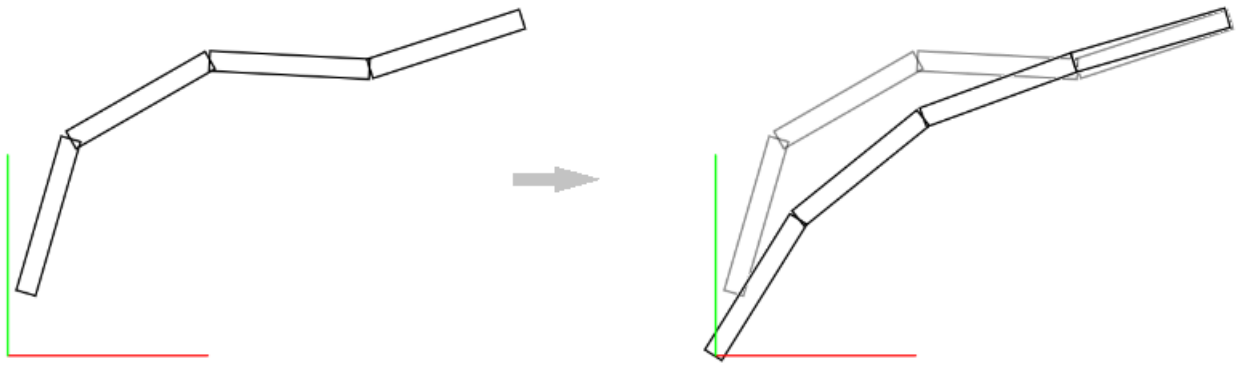

Figure 3.2: Kinematic Chain dragged to reach target followed by dragged backward towards base

3.1 Matlab Subroutine

```
%% Program to compute Inverse Kinematics of Multi link Planar Robot
%% Constants
clc; clear all; close all;
N=10;
baseXY =[0,0];
targetXY =[9,7];
%% Create Links
for i=1:N
    % Link array consist of [x y length theta endX endY]
    LinkArray(i,:)=[0 0 1 0 0 0];
end
%% Compute Inverse Kinematics
% pass array to function
[theta,X,Y,error] = InverseKinematics(baseXY(1),baseXY(2),targetXY(1),
   targetXY(2),LinkArray);
%% Plot Results
fprintf('\n           Target \n');
fprintf('link     X       Y    theta\n');
for i=N:-1:1
    if(i<N)
        thetac(i)=-theta(i)+theta(i+1);
    else
        thetac(i)=theta(i);
    end
   fprintf('%3i %6.2f %6.2f %6.2f deg\n',N-i+1,X(i), Y(i), thetac(i)*180/pi);
end
plot(X,Y,'-ok','LineWidth',4);
hold on;
plot(X(1),Y(1),'o','LineWidth',8);
plot(X(end),Y(end),'o','LineWidth',8);
legend('Links','End Effector','Base');
grid minor; axis equal;

figure
semilogy(error,'k-');
hold on;
semilogy(error,'r*');
title('Iteration vs Error');
xlabel('Iteration'); ylabel('error');
```

Code 3.1: Test program for 10 Links Target(5,7),Base(0,0)

```
function [theta,X,Y,err] = InverseKinematics(baseX,baseY,targetX,targetY,
    LinkArray)
%% function computes required theta for Planar Links given target and base
    cordinates
% returns array of required angles each represented with reference to base
    axis
N=size(LinkArray,1);
%MaxIter=50; %Maximum itterations if target is out of reach
tollerance = 1e-8;
%% Generate Random data
for i=2:N
    l=LinkArray(i,3);
    x=LinkArray(i-1,5);
    y=LinkArray(i-1,6);
    theta=LinkArray(i,4);
    bx=x+l*cos(theta);
    by=y+l*sin(theta);
    LinkArray(i,:)=[x,y,l,theta,bx,by];
end

%% loop till error > tollerance
itr=1;
error=sqrt((targetX-LinkArray(1,5))^2+(targetY-LinkArray(1,6))^2);
err(itr)=error; % store errors
%while((error>tollerance) && (j<=MaxIter))
prev_error=0;
while(abs(error-prev_error) > tollerance)
    prev_error=error;
    err(itr)=error;
    %% Drag towards Target
    LinkArray=DragForward(LinkArray,targetX,targetY);
    %% Drag towards Fixed Base
    LinkArray=DragBackward(LinkArray,baseX,baseY)

    error=sqrt((targetX-LinkArray(1,5))^2+(targetY-LinkArray(1,6))^2);
    itr=itr+1;
end
fprintf("Solution converged in %d itterations",itr);
%Output Arguments
X=[LinkArray(1,5);LinkArray(:,1)];
Y=[LinkArray(1,6);LinkArray(:,2)];
theta=LinkArray(:,4);
end
```

Code 3.2: Function to compute inverse kinematic solution from provided link configuration

```
function LinkArray = DragForward(LinkArray,targetX,targetY)
N=size(LinkArray,1);
    for i=1:N
        l=LinkArray(i,3);
        if(i==1)
            x=LinkArray(i,1);
            y=LinkArray(i,2);
            angle=atan2((y-targetY),(x-targetX));
            bx=targetX;
            by=targetY;
            x=targetX+l*cos(angle);
            y=targetY+l*sin(angle);
            LinkArray(i,:)=[x,y,l,angle,bx,by];
        else
            x=LinkArray(i,1);
            y=LinkArray(i,2);
            angle=atan2((y-LinkArray(i-1,2)),(x-LinkArray(i-1,1)));
            bx=LinkArray(i-1,1);
            by=LinkArray(i-1,2);
            x=bx+l*cos(angle);
            y=by+l*sin(angle);
            LinkArray(i,:)=[x,y,l,angle,bx,by];
        end
    end
end
```

Code 3.3: Forward Reaching subroutine

```
function LinkArray = DragBackward(LinkArray,baseX,baseY)
N=size(LinkArray,1);
    for i=N:-1:1
        l=LinkArray(i,3);
        if(i==N)
            bx=LinkArray(i,5);
            by=LinkArray(i,6);
            angle=atan2((by-baseY),(bx-baseX));
            x=baseX;
            y=baseY;
            bx=baseX+l*cos(angle);
            by=baseY+l*sin(angle);
            LinkArray(i,:)=[x,y,l,angle,bx,by];
        else
            bx=LinkArray(i,5);
            by=LinkArray(i,6);
            angle=atan2((by-LinkArray(i+1,6)),(bx-LinkArray(i+1,5)));
            x=LinkArray(i+1,5);
            y=LinkArray(i+1,6);
            bx=x+l*cos(angle);
            by=y+l*sin(angle);
            LinkArray(i,:)=[x,y,l,angle,bx,by];
        end
    end
end
```

Code 3.4: Backward Reaching subroutine

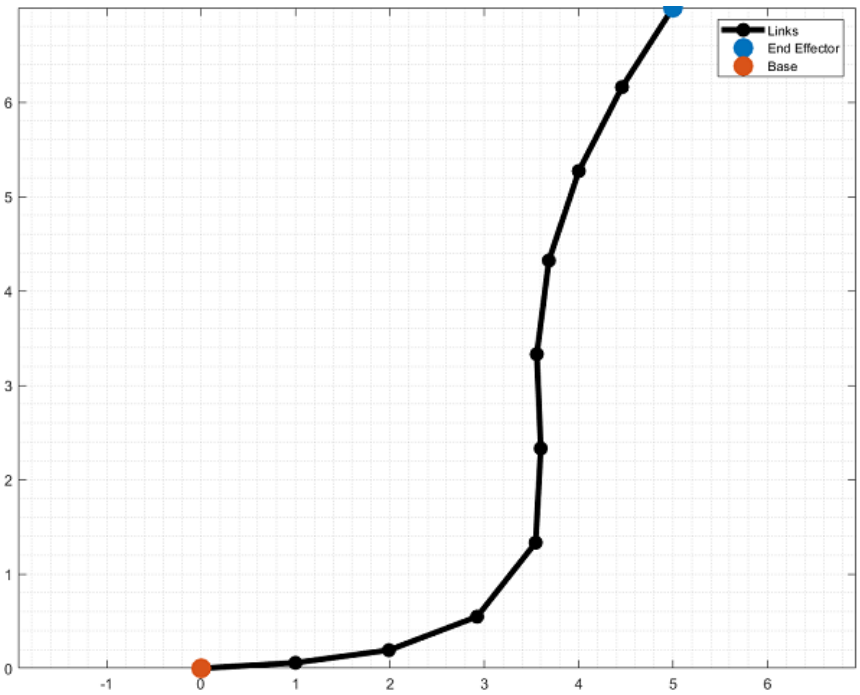

Figure 3.3: Solution Arm Configuration for 10 links, from Base(0,0) to Target (5,7)

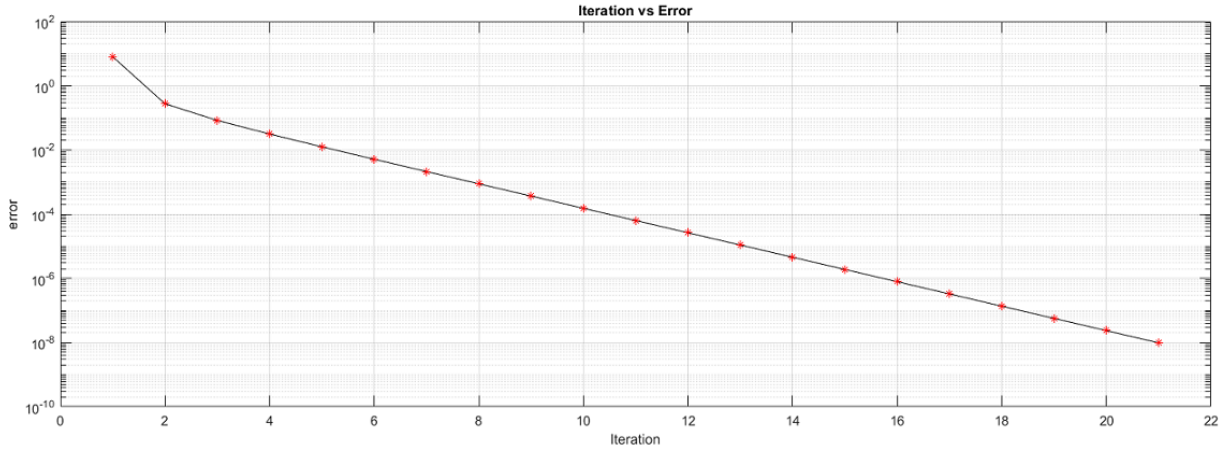

Figure 3.4: Error convergence for 10 Links, from Base(0,0) to Target (5,7)

We can notice convergence of of numerical scheme in Figure 3.4, within first five iteration error reduced below 10^{-2} units.

3.2 Realtime 3D Implementation

Although complexity of problem increased by from transition of two dimensional world to three dimension, the core idea behind numerical scheme is still same. Even in this case we rotate each individual link towards goal and all previous links in kinematic chain follows very next link.

First we need to find angle of rotation of link from current position towards target. Assuming all joints are spherical,rotation can be represented with only two angles namely **Azimuth** and **Elevation** often called as look angles. In our inertial reference system azimuth is rotation ψ around z-axis and elevation is rotation θ around y axis. Consider $A(x_i, y_i, z_i)$ is location of joint $B(x_i, y_i, z_i)$ is other end located at distance l_i from $A(x_i, y_i, z_i)$ and target coordinates are $T(x_i, y_i, z_i)$. Link is rotated at A towards target T The rotations can be computed from:

$$\Delta x = x_{target_i} - x_{A_i} \quad \Delta y = y_{target_i} - y_{A_i} \quad \Delta z = z_{target_i} - z_{A_i}$$

$$\psi = tan^{-1}\left(\frac{\Delta y}{\Delta x}\right) \tag{3.2}$$

$$\theta = tan^{-1}\left(\frac{\Delta z\, cos(\psi)}{\Delta x}\right) \tag{3.3}$$

After computing rotation link's B_i point is moved to target (in case of end effector/last link) coordinate keeping orientation angles θ_i and ψ_i constant. Similarly for subsequent links $i-1$ target is A_i and links are rotated at joint A_{i-1} then other end of link which is B_{i-1} is moved to A_i. In this way we get forward reaching chain with one end free and front end (end effector) is reached at goal.

To get full solution we need to drag entire chain backwards towards fixed base. We start from one end of first link A_i and rotate link i towards base by angle ψ_i and θ_i at joint B_i and A_i is moved to target keeping rotation angles same as computed. All next link follows there previous links target for link $i+1$ is B_i and rotation is at joint B_{i+1}.
Both forward scheme and backward scheme should be iterated one after another till acceptable error tolerance. Following listing 3.5 is algorithm to compute inverse kinematin solution for any generic open chain configuration.

```
while error > tolerance do
for i= N ... 1 do
    if i is N (Check if end effector)
        rotate link i at joint A towards target
        move point B to target
    else
        rotate link i at joint A towards joint A of link i+1
        move joint B of i to joint A of i+1
    end
end
for i= 1 ... N do
    if i is 1
        rotate link i at joint B towards base
        move joint A to base
    else
        rotate link i at joint B towards joint B of link i-1
        move joint A of i to joint B of i-1
    end
end
error = distance between target and end-effector
end
```

Code 3.5: Algorithm to compute inverse kinematic

Purpose of this exercise is to create an interactive simulation in which user can simulate change in configuration of robotic arm. It is important to have a smaller footprint package which is platform independent and produce same performance across all platform. The JavaScript programming being one of the oldest high level interpreted prototype based programming language is core of almost all web browsers. Although JavaScript was developed taking focusing on web browsers, nowadays we can see it everywhere due to popularity of cloud computing. Randall Munroe precisely point out current state of software paradigm in his webcomic XKCD.

Figure 3.5: XKCD on MAC/PC and browser based software

Javascript has huge developer community and has lots of open source libraries developed which provides simpler abstract yet powerful tools to interact with graphics such as Processing graphics library and interactive development environment. Ben Fry in his book Visualizing Data [4] and Daniel Shiffman in his bool The nature of code [8] explained various tools and techniques which are extensively used to solve this exercise.

Link object constructor shown in Code 3.6, takes initial coordinate of joint, length and diameter of link and returns object of type `Link`.

```
function Link(x,y,z,length,dia) {
    // joint coordinates of link
    this.x=x;
    this.y=y;
    this.z=z;
    this.length=length;
    // end coordinates link
    this.bx=0;
    this.by=0;
    this.bz=0;
    // link diameter
    this.dia=dia;
    // rotation angle of spherical joint
    this.rx=0;
    this.ry=0;
    this.rz=0;
}
```

Code 3.6: Link object constructor

Now The JavaScript prototype property also allows you to add new methods to objects constructors. `dragForward` and `dragBackword` is added to Link Object.

```
Link.prototype.dragForward = function(xx,yy,zz) {
    this.bx=xx;
    this.by=yy;
    this.bz=zz;
    // computed rotation angles psi and theta between two points
    let rotation = getDirection(this.bx,this.by,this.bz,this.x,this.y,this.z);

    r11=cos(rotation.z)*cos(rotation.y);
    r21=(sin(rotation.z)*cos(rotation.y));
    r31=(-sin(rotation.y));
    this.x=this.bx + this.length*r11;
    this.y=this.by + this.length*r21;
    this.z=this.bz + this.length*r31;
};
```

Code 3.7: Drag Forward function (for Forward Reaching)

```
Link.prototype.dragBackword = function(xx,yy,zz) {
  this.x=xx;
  this.y=yy;
  this.z=zz;
  // computed rotation angles psi and theta between two points
  let rotation = getDirection(this.bx,this.by,this.bz,this.x,this.y,this.z);

  r11=cos(rotation.z)*cos(rotation.y);
  r21=(sin(rotation.z)*cos(rotation.y));
  r31=(-sin(rotation.y));
  this.bx=this.x-this.length*r11;
  this.by=this.y-this.length*r21;
  this.bz=this.z-this.length*r31;
  this.rx=rotation.x;
  this.ry=rotation.y;
  this.rz=rotation.z;
};
```

Code 3.8: Drag Backward Function (for backward Reaching)

```
function getDirection(x1,y1,z1,x2,y2,z2) {
    let dx=x2-x1; // x distance
    let dy=y2-y1; // y distance
    let dz=z2-z1; // z distance
    let len = sqrt(dx*dx + dy*dy + dz*dz); // resultant length
    let rotx = 0;
    let roty = 0;
    let rotz = Math.atan2( dy, dx );
    if(dx>=0) {
        roty = -Math.atan2(dz*Math.cos(rotz),dx); // avoid numerical
            singularity
    }
    else {
        roty = Math.atan2(dz*Math.cos(rotz),-dx);
    }

    let rotation= {'x':rotx,'y':roty,'z':rotz,'r':len};
    return rotation;
}
```

Code 3.9: Function to compute rotation angles from two coordinates)

Object of type `Link` which has information about link of robot is pushed to `link_array`. Now method `dragForward` is called for first link with target coordinates as argument. Further `dragForward` is called all subsequent links from $i = 1$ to $i = N - 1$ with coordinates of link $i - 1$ as argument.

For backward reaching link $i = N - 1$ is `dragForward` with base coordinates as argument followed by links $i = N - 2$ to $i = 0$ is `dragForward` with end coordinates of link $i + 1$. Finally, `link_array` consist of new updated solution to reach desired goal.

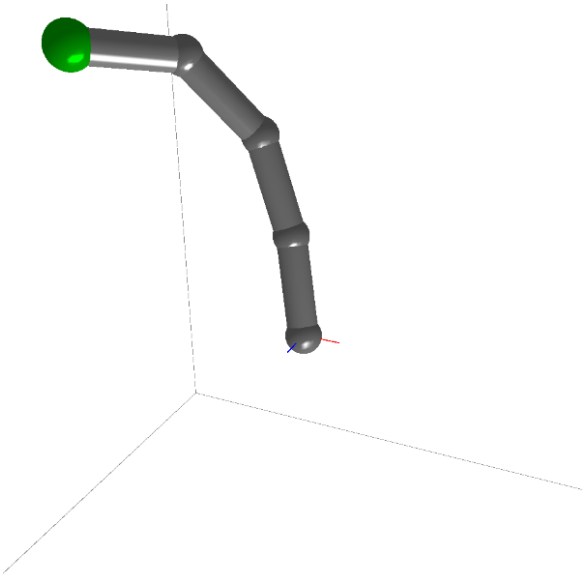

Figure 3.6: Robot reached to desired goal (Green Ball)

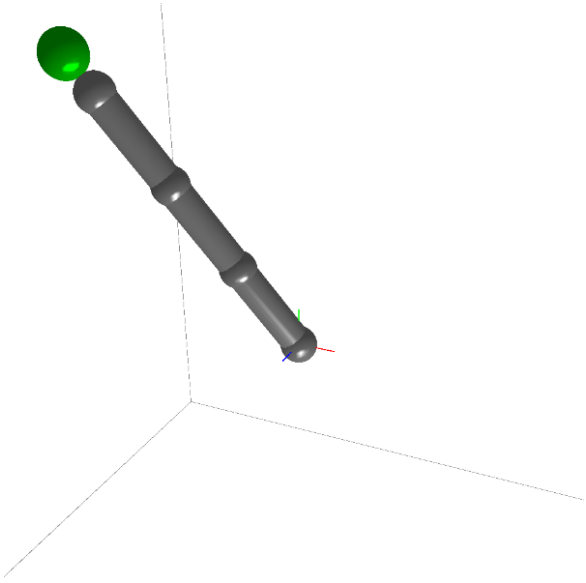

Figure 3.7: After removing one link in real time

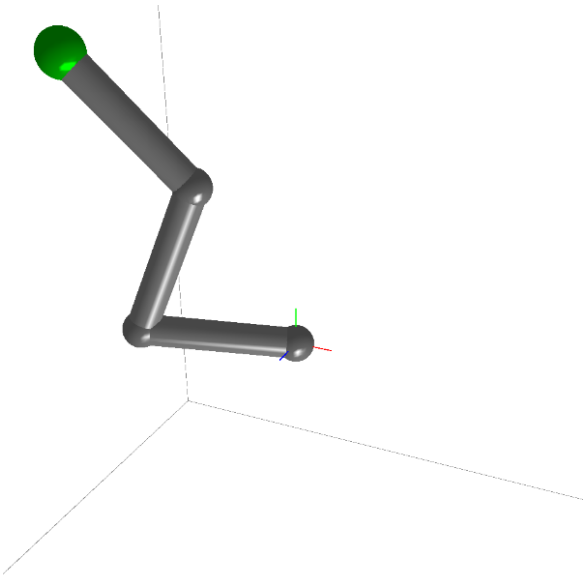

Figure 3.8: Link added in real-time

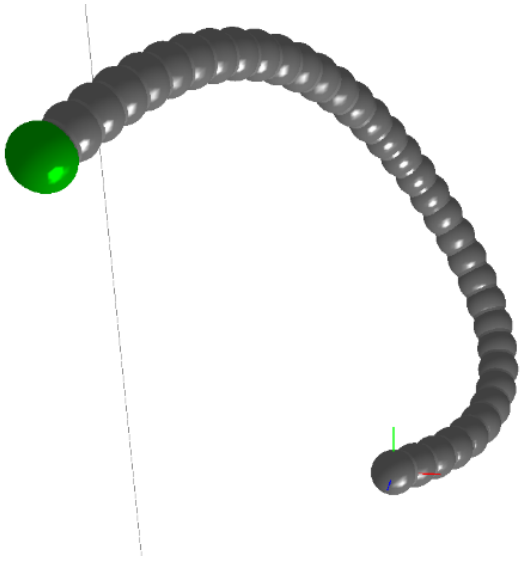

Figure 3.9: Caterpillar, tentacle or Tesla Charging Arm?

Figure 3.8 shows robot reached at goal, after removing one link in real-time shown in Figure 3.6 arm is trying to reach target out of it's work volume. Further one link is added again now Figure 3.7 shows goal is achieved. We can notice that converged solution depends on previous configuration and robot finds shortest path from current position to desired target.

3.3 Application

FABRIK algorithm could be good heuristic for Artificial Intelligence Search problems of kinematic chain. Character animation in games and simulations needs real-time solution for inverse kinematics, with this scheme movement of limbs are similar to natural moment of arm since it finds shortest possible route towards goal.

Another possible application could be hypothetical scenario of a humanoid robot whose arm has grabbed some tool or stick which is not already modeled in configuration of robot. Now robots virtual work volume is increased considering grabbed object as a link.

A quadruped robot walking on rough terrain could use this heuristic to find joint angles for specific arm motion in real time.

Tentacles of Bio mimic creature, like Tesla automated charging arm.

3.3.1 Future Scope

Numerical solution is found assuming all joints are spherical revolute and has scope to add prismatic joints. Constraints shall be implemented to provide more realistic results.

3.3.2 Source Code and Demo

1. 3D Demo - `http://siroi.co.in/apps/InverseKinematics3D/`
2. 2D Demo - `http://siroi.co.in/apps/FABRIK2D/`
3. Sourece Code - `https://github.com/siddharthdeore/InverseKinematics`

List of Figures

List of Codes

Bibliography

[1] Andreas Aristidou and Joan Lasenby. FABRIK: A fast, iterative solver for the inverse kinematics problem. *Graph. Models*, 73(5):243–260, September 2011.

[2] Joel Brown, Jean-Claude Latombe, and Kevin Montgomery. Real-time knot-tying simulation. *The Visual Computer*, 20(2-3):165–179, 2004.

[3] Samuel R Buss. Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. *IEEE Journal of Robotics and Automation*, 17(1-19):16, 2004.

[4] B. Fry. *Visualizing Data: Exploring and Explaining Data with the Processing Environment*. Data Analysis. O'Reilly Media, Incorporated, 2008.

[5] Prof. Alessandro De Luca. Robotics 1 - inverse kinematics, accesed December 5, 2018. `http://www.diag.uniroma1.it/~deluca/rob1_en/10_InverseKinematics.pdf`.

[6] Michael Meredith and Steve Maddock. Real-time inverse kinematics: The return of the jacobian. Technical report.

[7] R Muller-Cajar and R Mukundan. Triangualation-a new algorithm for inverse kinematics. 2007.

[8] D. Shiffman, S. Fry, and Z. Marsh. *The Nature of Code*. D. Shiffman, 2012.

[9] C. W. Wampler. Manipulator inverse kinematic solutions based on vector formulations and damped least-squares methods. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):93–101, Jan 1986.

[10] L-CT Wang and Chih-Cheng Chen. A combined optimization method for solving the inverse kinematics problems of mechanical manipulators. *IEEE Transactions on Robotics and Automation*, 7(4):489–499, 1991.