# SYSM 6305 Optimization Theory and Practice
# Fall 2019

# Project Report

The goal with forward kinematics is to find the location/orientation of the end effector of an arm with joint angles + displacement as input. The goal with inverse kinematics is to take the desired location/orientation of the end effector as input and find the joint angles + displacements needed to get to this location.

Inverse kinematics is more complicated than forward kinematics due to the possibility of multiple solution or no solutions. An example of multiple solutions would be an arm with extra degrees of freedom being able to get to the same orientation in multiple different ways. Some examples of no solutions would be a position that is far away from the arm and therefore out of reach, or a position that the arm cannot get to without colliding with itself.

Jacobian Matrix: The Jacobian matrix is a matrix of all the first order partial derivatives of a vector function. This means that the Jacobian matrix describe how each coordinate changes with respect to each joint angle in our system. It describes the first order linear approximation between these arguments.

Methods Used for the optimization problem:

1. Newton Raphson
2. Gradient Descent

Comparison between Newton Raphson and Gradient Descent:

| Sl No. | Learning Rate | Newton Raphson Iteration | Gradient-Descent Iterations |
|---|---|---|---|
| 1. | 0.5 | 1100 | 3295 |
| 2 | 2.5 | 221 | 658 |
| 3 | 5.5 | 101 | 298 |
| 4 | 8.5 | 70 | 192 |
| 5 | 12.5 | 44 | 130 |
| 6 | 15 | 37 | 108 |
| 7 | 20 | 31 | 81 |
| 8 | 25 | 26 | 64 |
| 9 | 30 | 18 | 52 |
| 10 | 35 | 19 | 45 |

**Conclusion:** We can see clearly as learning rate is increasing, number of iterations are decreasing to some point and then again start to increase. Also, Newton Raphson takes way less Iterations to converge to a minimal.

## Code Base for Newton Raphson:

```
%% Function instructions:
% Put:
% a1,a2 : Robot arm lenght
% X1d,X2d: End effector points
% theta_1,theta_2: Starting point angle
% rate: Learning rate
% call function: [count,start_eff,cost] = Newton_Raphson(1,1,.1,1.5,0.2,0.6,30)
function [count,start_eff,cost] = Newton_Raphson(a1,a2,X1d,X2d, theta_1, theta_2,rate)
% Counter for number of Iterations
count = 0;

% Calculate Initial starting points using Forward Kinematics
X1 = a1*cosd(theta_1) + a2*cosd(theta_1+theta_2);
X2 = a1*sind(theta_1) + a2*sind(theta_1+theta_2);

% Finding Jacobian Matrix
Jacobian = [-a1*sind(theta_1)-a2*sind(theta_1+theta_2), -a2*sind(theta_1+theta_2);
            a1*cosd(theta_1)+a2*cosd(theta_1+theta_2), a2*cosd(theta_1+theta_2)];

O = [0,0]
% Vector matrix of desired location of robotic arm
end_eff = [X1d;X2d];

% Vector matrix of starting point of robotic arm
start_eff = [X1;X2];

% Vector matrix of starting potint of robotic arm
theta_eff = [theta_1;theta_2];
```

```
% Error related to end effector and starting point
error = start_eff-end_eff;

% Cost function based on euclidean distance of end effector and starting point
cost = sqrt((end_eff(1,1)-start_eff(1,1))^2+(end_eff(2,1)-start_eff(2,1))^2);

% Iterating the process and decreasing the cost to reach end end effector
while (cost > 0.0001)
    end_eff = [X1d;X2d]
    X1 = a1*cosd(theta_eff(1,1)) + a2*cosd(theta_eff(1,1)+theta_eff(2,1))
    X2 = a1*sind(theta_eff(1,1)) + a2*sind(theta_eff(1,1)+theta_eff(2,1))
    Jacobian = [-a1*sind(theta_eff(1,1))-a2*sind(theta_eff(1,1)+theta_eff(2,1)), -a2*sind(theta_eff(1,1)+theta_eff(2,1));
            a1*cosd(theta_eff(1,1))+a2*cosd(theta_eff(1,1)+theta_eff(2,1)), a2*cosd(theta_eff(1,1)+theta_eff(2,1))]
    start_eff = [X1;X2]% f(q)
    error = start_eff-end_eff % f(q) - xd
    inv_jacob = inv(Jacobian)
    theta_eff = theta_eff - rate*(inv_jacob)*error % qk+1 = qk-lr*inv(j)*(f(q)-xd)
    cost = sqrt((end_eff(1,1)-start_eff(1,1))^2+(end_eff(2,1)-start_eff(2,1))^2) % Euclidean distance
    count = count + 1 % Counting Iterations

    % Visualization %
    P=a1*[cosd(theta_eff(1,1)) sind(theta_eff(1,1))]
    Q=[P(1)+a2*cosd(theta_eff(1,1)+theta_eff(2,1)) P(2)+a2*sind(theta_eff(1,1) + theta_eff(2,1))]
    O_circ = viscircles(O,0.03,'Color', 'k','LineWidth',2)
    link_1 = line([O(1) P(1)], [O(2) P(2)], 'Color','r','LineWidth',1)
    ball = viscircles(P,0.03,'Color','k','LineWidth',3)
    link_2 = line([P(1) Q(1)], [P(2) Q(2)],'Color','b','LineWidth',1)

end
```

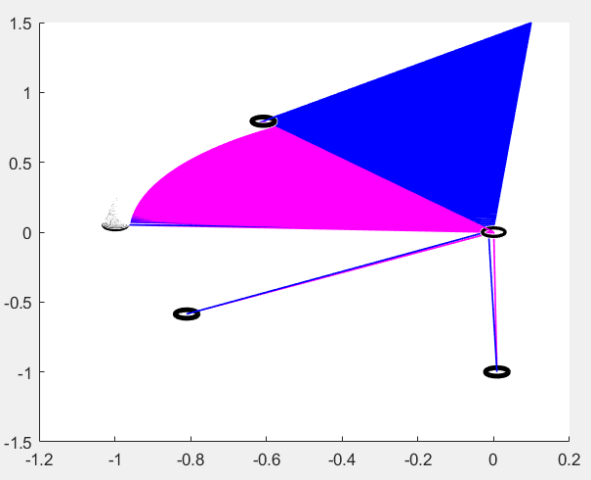
## Code Base for Gradient Descent:

```
%% Function instructions:
% Put:
% a1,a2 : Robot arm lenght
% X1d,X2d: End effector points
% theta_1,theta_2: Starting point angle
% rate: Learning rate
% call function: [count,start_eff,cost] = Gradient_Descent(1,1,.1,1.5, 0.2,0.6,30)
function [count,start_eff,cost] = Gradient_Descent(a1,a2,X1d,X2d,theta_1,theta_2,rate)
% Counter for number of Iterations
count = 0;

% Calculate Initial starting points using Forward Kinematics
X1 = a1*cosd(theta_1) + a2*cosd(theta_1+theta_2);
X2 = a1*sind(theta_1) + a2*sind(theta_1+theta_2);

% Finding Jacobian Matrix
Jacobian = [-a1*sind(theta_1)-a2*sind(theta_1+theta_2), -a2*sind(theta_1+theta_2);
            a1*cosd(theta_1)+a2*cosd(theta_1+theta_2), a2*cosd(theta_1+theta_2)];

% Vector matrix of desired location of robotic arm
end_eff = [X1d;X2d];
O = [0,0]
% Vector matrix of starting point of robotic arm
start_eff = [X1;X2];

% Vector matrix of starting potint angle of robotic arm
theta_eff = [theta_1;theta_2];
```

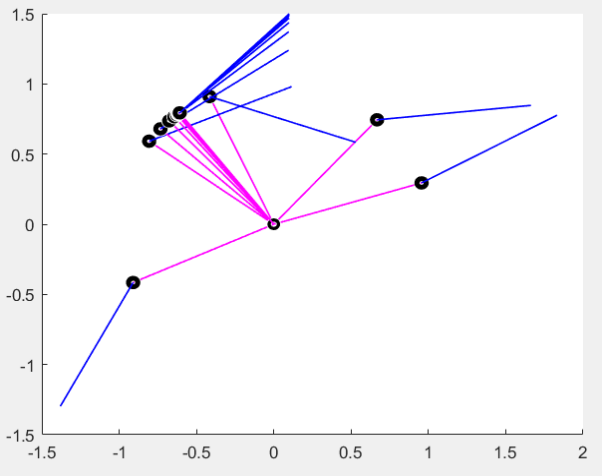
```
% Error related to end effector and starting point
error = start_eff-end_eff;

% Cost function based on euclidean distance of end effector and starting point
cost = sqrt((end_eff(1,1)-start_eff(1,1))^2+(end_eff(2,1)-start_eff(2,1))^2);

% Iterating the process and decreasing the cost to reach end end effector
while (cost > 0.0001)
    end_eff = [X1d;X2d]
    X1 = a1*cosd(theta_eff(1,1)) + a2*cosd(theta_eff(1,1) + theta_eff(2,1))
    X2 = a1*sind(theta_eff(1,1)) + a2*sind(theta_eff(1,1) + theta_eff(2,1))
    Jacobian = [-a1*sind(theta_eff(1,1))- a2*sind(theta_eff(1,1)+ theta_eff(2,1)), -a2*sind(theta_eff(1,1)+ theta_eff(2,1));
            a1*cosd(theta_eff(1,1)) + a2*cosd(theta_eff(1,1) + theta_eff(2,1)), a2*cosd(theta_eff(1,1) + theta_eff(2,1))]
    start_eff = [X1;X2]% f(q)
    error = start_eff - end_eff % f(q) - xd
    trans_jacob = transpose(Jacobian)% transpose of Jacobian
    theta_eff = theta_eff - (rate)*(trans_jacob)*(error)  % qk+1 = qk-lr*transpose(j)*(f(q)-xd)
    cost = sqrt((end_eff(1,1)-start_eff(1,1))^2+(end_eff(2,1)-start_eff(2,1))^2) % Euclidean distance
    count = count + 1 % Counting Iterations
    % Visualization %
    P=a1*[cosd(theta_eff(1,1)) sind(theta_eff(1,1))]
    Q=[P(1)+a2*cosd(theta_eff(1,1)+theta_eff(2,1)) P(2)+a2*sind(theta_eff(1,1) + theta_eff(2,1))]
    O_circ = viscircles(O,0.03,'Color', 'k','LineWidth',2)
    link_1 = line([O(1) P(1)], [O(2) P(2)], 'Color','m','LineWidth',1)
    ball = viscircles(P,0.03,'Color','k','LineWidth',3)
    link_2 = line([P(1) Q(1)], [P(2) Q(2)],'Color','b','LineWidth',1)
end
```

## Visualizations:

1. Newton Raphson visualizations for Learning rate [0.5 and 30]

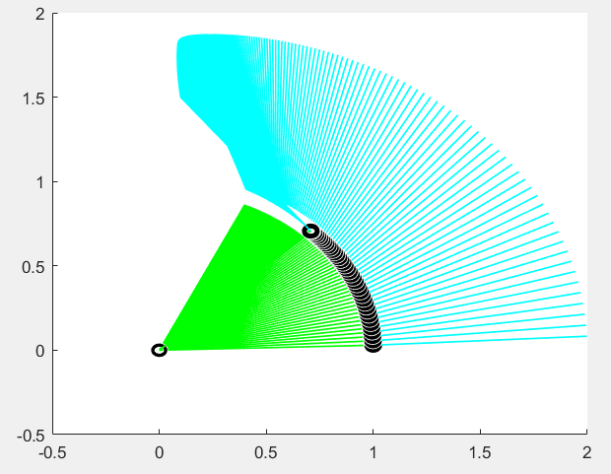



**Learning Rate: 0.5, Iterations: 1100**

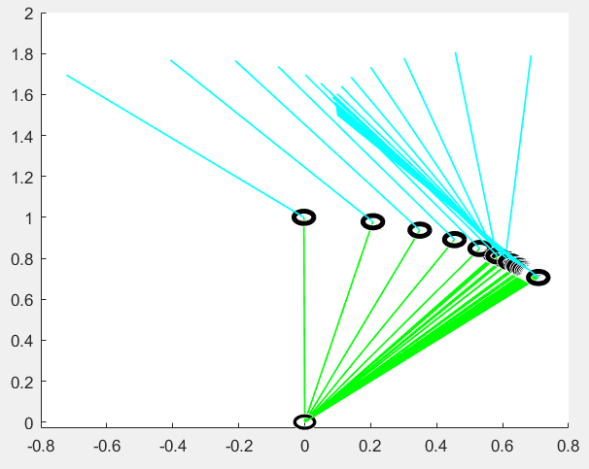



**Learning Rate: 30, Iterations: 18**

2. Gradient Descent visualizations for Learning rate [0.5 and 30]




**Learning Rate: 0.5, Iterations: 3295**




**Learning Rate: 30, Iterations: 52**