Arpit Savarkar
==

Q7 :

```
import numpy as np
import matplotlib.pyplot as plt
import math
import enum

"""

Kinematic Analysis of 3 link robot is undertaken using Newton Raphson method of Inverse Jacobian

The following, Code excerpt was read and followed from the book " Modern Robotics - Motion Planning and Contr
ol "

 Assume thatf:Rn→Rm is differentiable, and letxdbe the desired end-effector coordinates.
 Theeta(θ)for the Newton-Raphson method is defined as g(θ) = xd−f(θ), and the goal is to find joint coordinates θd s
uch that
 g(θd) = xd − f(θd) = 0.

 Given an initial guessθ0 which is “close to” a solution θd, the kinematics can be expressed as the Taylor expansion.
 Ignoring the Higher Order Terms, The Jacobian forms, Update Step leads to convergence.

 The first step of the Newton–Raphson method for nonlinear root-finding for  a  scalar x and θ.
 In  the  first  step,  the  slope −∂f/∂θ is  evaluated  at  the  point(θ0,xd−f(θ0)).
 In the second step, the slope is evaluated at the point (θ1,xd−f(θ1)) and eventually the process converges to θd.
 Note that an initial guess to the left of the plateau of xd−f(θ) would be likely to result in convergence to the other ro
ot of
 xd−f(θ), and an initial guess at or near the plateau would result in a large initial |∆θ| and the iterative process might
not
 converge at all.

 PseudoInverse needs to be used, since this is not a square matrix.
 Replacing the Jacobian inverse with the pseudoinverse, ∆θ = J†(θ0) (xd−f(θ0))

 The Following Pseudo Code explains the overview,
 Newton–Raphson iterative algorithm for finding θd:
 (a) Initialization:  Given xd□ Rm and an initial guess θ0□ Rn, set
    i= 0.
 (b) Set e=xd−f(θi).  While epsilon for some small theta : Set θi+1=θi+J†(θi)e.
 ->Incrementi

"""

class State_Machine(enum.Enum):
  UPDATE_GOAL = 1
  MOVE = 2

class Robot(object):
    """
    This Class is helper class for plotting and mainipulating which keeps track the end points.
```

```
@param - arm_lengths : Array of the Lengths of each arm
@param - motor_angles : Current Angle of the Each Revolute Joint in the Global Frame
@goal  - Final Position expected to be reached by the link
"""
def __init__(self, arm_lengths, motor_angles, goal):
    """
    Initialization
    """
    self.arm_lengths = np.array(arm_lengths)
    self.motor_angles = np.array(motor_angles)
    self.link_end_pts = [[0, 0], [0, 0], [0, 0], [0, 0]]
    self.goal = np.array(goal).T
    self.lim = sum(arm_lengths)
    plt.ion()
    plt.show()
    # Find the Location of End Points of each Link
    for i in range(1, 4):
        self.link_end_pts[i][0] = self.link_end_pts[i-1][0] + self.arm_lengths[i-1] * \
            np.cos(np.sum(self.motor_angles[:i]))
        self.link_end_pts[i][1] = self.link_end_pts[i-1][1] + self.arm_lengths[i-1] * \
            np.sin(np.sum(self.motor_angles[:i]))

    # Explicity Setting The end effector Position
    self.end_effector = np.array(self.link_end_pts[3]).T
    self.plot()

def update_joints(self, motor_angles):
    """
    Update the Location of the end points of the link, Based on Updates of the End points
    """
    self.motor_angles = motor_angles

    # Update Steps
    # Set e=xd−f(θi).  While epsilon for some small theta : Set θi+1=θi+J†(θi)e.
    for i in range(1, 4):
        # Cosine length Update
        self.link_end_pts[i][0] = self.link_end_pts[i-1][0] + self.arm_lengths[i-1] * \
            np.cos(np.sum(self.motor_angles[:i]))
        # Sine length Update
        self.link_end_pts[i][1] = self.link_end_pts[i-1][1] + self.arm_lengths[i-1] * \
            np.sin(np.sum(self.motor_angles[:i]))

    # Explicity Setting The end effector Position
    self.end_effector = np.array(self.link_end_pts[3]).T
    self.plot()

def plot(self):
    """
    Helper functions to plot links, Motor joints, based on Newton Raphson Jacobian Inverse Calculation
    """
    plt.cla()
    for i in range(4):
        if i is not 3:
            # Plot Links
```

```
            plt.plot([self.link_end_pts[i][0], self.link_end_pts[i+1][0]],\
                [self.link_end_pts[i][1], self.link_end_pts[i+1][1]], 'c-')
        # Plot Motor Joint
        plt.plot(self.link_end_pts[i][0], self.link_end_pts[i][1], 'ko')

    # Mark the goal Position
    plt.plot(self.goal[0], self.goal[1], 'rx')
    plt.xlim([-self.lim, self.lim])
    plt.ylim([-self.lim, self.lim])
    plt.draw()
    plt.pause(0.0001)


def inv_K(arm_lengths, motor_angles, goal):
  """
  Inverse Kinematics for Analysis to calculate Jacobian, to update the non-linear equations ignoring the higher orde
r terms
  """
  # Number of Iterations here is 30000,
  for itr in range(30000):
    J = np.zeros((2, 3))
    # Calculates the Forward Kineamatics of the robot for the current transform
    transform_t = forw_K(arm_lengths, motor_angles)
    epsilon, distance = np.array([(goal[0] - transform_t[0]), (goal[1] - transform_t[1])]).T,\
        np.hypot((goal[0] - transform_t[0]), (goal[1] - transform_t[1]))

    # Success Condition
    if distance < 1:
      return motor_angles, True

    # Update Jacobian
    for i in range(3):
      J[0, i] = J[1, i] = 0
      for j in range(i, 3):
        J[0, i] -= arm_lengths[j] * np.sin(np.sum(motor_angles[:j]))
        J[1, i] += arm_lengths[j] * np.cos(np.sum(motor_angles[:j]))

    # Angle Update Step
    # θi+1=θi+J†(θi)e.
    motor_angles = motor_angles + np.dot(np.linalg.pinv(J), epsilon)
  return motor_angles, False


def forw_K(arm_lengths, motor_angles):
  """
  Function to Calculate the forward kinematics.
  """
  pos_x = pos_y = 0
  # Simple logic gets the calculates the End Effector position
  # from the current motor angle and position
  for i in range(1, 4):
    pos_x += arm_lengths[i-1] * np.cos(np.sum(motor_angles[:i]))
    pos_y += arm_lengths[i-1] * np.sin(np.sum(motor_angles[:i]))
```

```
    # Transpose is necessary, for future updates
    return np.array([pos_x, pos_y]).T


def main():
    """
    Main functionalities
    """

    # Length Setup
    l1 = input("Enter Length of Link 1\n")
    l2 = input("Enter Length of Link 2\n")
    l3 = input("Enter Length of Link 3\n")
    arm_lengths = [float(l1), float(l2), float(l3)]

    # Default Values
    motor_angles = np.array([np.radians(30)] * 3)
    goal_pos = [0.1, 4.1]
    output = input("Do you want to Enter angles (Enter 'angle') or Enter final goal(Enter 'goal')")

    # Calculates the final Goal from angle
    if(output == 'angle'):
        motor_angle1 = input("Enter Angle of Link 1\n")
        motor_angle2 = input("Enter Angle of Link 2\n")
        motor_angle3 = input("Enter Angle of Link 3\n")
        motor_angles = np.radians(np.array([float(motor_angle1), float(motor_angle2), float(motor_angle3)]))
        print("Wanted Motor Angles")
        print(np.degrees(motor_angles))
        final_goal_pos = forw_K(arm_lengths, motor_angles)
        print("End Effector Position")
        print(final_goal_pos)

    # For inverse kinematics
    elif (output == 'goal'):
        goal_x = input("Enter X Coordinate of Goal\n")
        goal_y = input("Enter Y Coordinate of Goal\n")
        final_goal_pos = [float(goal_x), float(goal_y)]

    # Object of concern
    arm = Robot(arm_lengths, motor_angles, goal_pos)
    # Initializes the State to some default
    state = State_Machine.UPDATE_GOAL
    solution_found = False

    # Helper Flag
    goal_counter = 0
    while True:
        # Helper functions
        old_goal = np.array(goal_pos)
        goal_pos = np.array(arm.goal)
        distance = np.hypot((goal_pos[0] - arm.end_effector[0]), (goal_pos[1] - arm.end_effector[1]))

        # Inspired from the concept of Embedded Systems which uses State Machine
        # To stay in an infinite connected loop
```

```
        if state is State_Machine.UPDATE_GOAL:
            # Sucess condition
            if distance > 0.1 and not solution_found:
                # Gives the Updates over convergence
                joint_goal_angles, solution_found = inv_K(arm_lengths, motor_angles, goal_pos)
                if not solution_found:
                    # Still Convergence Condition is not met
                    print("Goal Unreachable")
                    state = State_Machine.UPDATE_GOAL
                    arm.goal = final_goal_pos
                elif solution_found:
                    # Continue Updates
                    state = State_Machine.MOVE
                    arm.goal = final_goal_pos
                if distance < 0.1:
                    # Sucess Condition
                    print("Joint Angles")
                    np.radians(np.degrees(motor_angles))
                    break
        # Second State Machine,
        elif state is State_Machine.MOVE:
            # Motor Angle Updates
            if distance > 0.1 and all(old_goal == goal_pos):
                motor_angles = motor_angles + (2 * ((joint_goal_angles - motor_angles + np.pi) % (2 * np.pi) - np.pi) * 0
.01)
            else:
                # Update State Machine
                state = State_Machine.UPDATE_GOAL
                solution_found = False
                arm.goal = final_goal_pos
                goal_counter += 1

            if distance < 1:
                    print("Joint Angles")
                    print(np.degrees(np.asarray(motor_angles)))

        # Runs 5 iterations to goal counter for success
        if goal_counter >= 5:
            break

        # Jacobian Update
        arm.update_joints(motor_angles)


if __name__ == '__main__':
    main()
```

==
==============================
Q8:
==
==============================

```
from math import pi
```

```
import numpy as np
import matplotlib.pyplot as plt
import shapely.geometry as geom
import descartes

# Simulation parameters
limit = 200

class Robot(object):
    """
    This Class is helper class for plotting and mainipulating which keeps track the end points.
    @param - arm_lengths : Array of the Lengths of each arm
    @param - motor_angles : Current Angle of the Each Revolute Joint in the Global Frame
    """
    def __init__(self, arm_lengths, motor_angles):
        # Initialization with a specific parameter
        self.arm_lengths = np.array(arm_lengths)
        self.motor_angles = np.array(motor_angles)
        self.link_end_pts = [[0, 0], [0, 0], [0, 0]]
        # Find the Location of End Points of each Link
        for i in range(1, 3):
            # Follows Forward Kinematic Update Steps Analysis
            self.link_end_pts[i][0] = self.link_end_pts[i-1][0] + self.arm_lengths[i-1] * \
                np.cos(np.sum(self.motor_angles[:i]))
            self.link_end_pts[i][1] = self.link_end_pts[i-1][1] + self.arm_lengths[i-1] * \
                np.sin(np.sum(self.motor_angles[:i]))

        self.end_effector = np.array(self.link_end_pts[2]).T

    def update_joints(self, motor_angles):
        """
        Update the Location of the end points of the link, Based on Updates of the End points
        """
        self.motor_angles = motor_angles
        # Forward Kinematic Update and storage of link_length data
        for i in range(1, 3):
            self.link_end_pts[i][0] = self.link_end_pts[i-1][0] + self.arm_lengths[i-1] * \
                np.cos(np.sum(self.motor_angles[:i]))
            self.link_end_pts[i][1] = self.link_end_pts[i-1][1] + self.arm_lengths[i-1] * \
                np.sin(np.sum(self.motor_angles[:i]))

        self.end_effector = np.array(self.link_end_pts[2]).T



def main():
    """
    Code expects the user to input to Input the Number of obstacles, the number of vertex inside the obstacle
    and the coordinates of each obstacle.

    Sample Solution of the following is attached with the folder
    a)  a workspace with a triangular obstacle with vertices (0.25,0.25), (0,0.75), and (−0.25,0.25).
    b)  a workspace with two large rectangular obstacles with vertices:
        O1:   (−0.25,1.1),(−0.25,2),(0.25,2),and  (0.25,1.1),
```

```
    O2:   (−2,−2),(−2,−1.8),(2,−1.8),and  (2,−2).
c)  a workspace with two obstacles:
    O1:   (−0.25,1.1),(−0.25,2),(0.25,2),and  (0.25,1.1),
    O2:   (−2,−0.5),(−2,−0.3),(2,−0.3),and  (2,−0.5)

Ctrl+C is needed to exit the program
"""
polygons_list = list()
vertex_list = list()
l1 = input("Enter Length of Link 1\n")
l2 = input("Enter Length of Link 2\n")
arm_lengths = [float(l1), float(l2)]
motor_angles = np.array([0] * 2)


num_obs = int(input("Enter the number of obstacles: "))
assert num_obs > 0
for obs in range(num_obs):
    print("For Obstacle :", obs+1)
    num_vertex = int(input("Enter the number of Vertexes of Polygon: "))
    for v in range(num_vertex):
        print("For Vertex :", v+1)
        x = float(input("Enter the X coordinate of Vertex: "))
        y = float(input("Enter the Y coordinate of Vertex: "))
        vertex_list.append((x,y))
    polygons_list.append(geom.Polygon(vertex_list))
    vertex_list = []

obstacles = geom.MultiPolygon(polygons_list)

print("Plotting Configuration-Space")

plt.ion()
plt.show(block=False)
arm = Robot(arm_lengths, motor_angles)

#Subdivide the The plot in a 100 by 100 grid
grid = [[0 for _ in range(limit)] for _ in range(limit)]
theta_list = [2 * i * pi / limit for i in range(-limit // 2, limit // 2 + 1)]
for i in range(limit):
    for j in range(limit):
        # Rotates the 2 link robot in the
        arm.update_joints([theta_list[i], theta_list[j]])
        link_end_pts = arm.link_end_pts
        collision_detected = False
        for k in range(len(link_end_pts) - 1):
            for obstacle in obstacles:
                line_seg = [link_end_pts[k], link_end_pts[k + 1]]
                line = geom.LineString([link_end_pts[k], link_end_pts[k + 1]])
                collision_detected = line.intersects(obstacle)
                if collision_detected:
                    break
            if collision_detected:
                break
```

```
            grid[i][j] = int(collision_detected)
    plt.imshow(grid)
    plt.pause(100)
    plt.show()



if __name__ == '__main__':
    main()
```

==
==============================

Q5:

==
==============================

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial import ConvexHull, convex_hull_plot_2d
from math import sin, cos, radians
import scipy
import pylab
from mpl_toolkits.mplot3d import Axes3D



def rotate_point(point, angle, center_point=(0, 0)):
    """Rotates a point around center_point(origin by default)
    Angle is in degrees.
    Rotation is counter-clockwise
    """
    angle_rad = radians(angle % 360)
    # Shift the point so that center_point becomes the origin
    new_point = (point[0] - center_point[0], point[1] - center_point[1])
    new_point = (new_point[0] * cos(angle_rad) - new_point[1] * sin(angle_rad),
                 new_point[0] * sin(angle_rad) + new_point[1] * cos(angle_rad))
    # Reverse the shifting we have done
    new_point = (new_point[0] + center_point[0], new_point[1] + center_point[1])
    return new_point


def rotate_polygon(polygon, angle, center_point=(0, 0)):
    """Rotates the given polygon which consists of corners represented as (x,y)
    around center_point (origin by default)
    Rotation is counter-clockwise
    Angle is in degrees
    """
    rotated_polygon = []
    for corner in polygon:
        rotated_corner = rotate_point(corner, angle, center_point)
        rotated_polygon.append(rotated_corner)
    return rotated_polygon
```

```
def rotate_at_angle(robot, angle):
    centroid_x = sum([x[0] for x in robot])/3
    centroid_y = sum([y[1] for y in robot])/3
    return rotate_polygon(robot, angle, (centroid_x,centroid_y))


def minkowski_sum(obstacle, robot):
    ms = []
    res = []
    for i in range(len(obstacle)):
        for j in range(len(robot)):
            ms.append(( obstacle[i][0] + robot[j][0] , obstacle[i][1] + robot[j][1]))
    ms.sort()
    for pts in ms:
        if pts not in res:
            res.append(pts)
    return res


def main():

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    obstacle = [(0, 0), (0, 2), (1, 2)]
    levels = input("Enter Number of Levels to see in the 3d Plots between 0 and 360\n")
    robot = [(-1.0*x[0], -1.0*x[1]) for x in obstacle]
    r_val = np.linspace(0, 360, int(levels))
    final_CSpace = []
    for r in r_val:
        CSpace = minkowski_sum(obstacle, robot)
        points = np.array([*CSpace])
        hull = ConvexHull(points)
        for simplex in hull.simplices:
            ax.plot3D(points[simplex, 0], points[simplex, 1], r, 'b-')
        robot = rotate_at_angle(robot, r)

    plt.show()



if __name__ == '__main__':
    main()
```

==

Q2: -> Matlab Analysis

==

```
>> syms alpha
>> syms beta
>> syms gamma
>>
>> RzGamma = [ cos(gamma) -sin(gamma) 0; sin(gamma) cos(gamma) 0; 0 0 1]
```

```
RzGamma =

[cos(gamma), -sin(gamma), 0]
[sin(gamma),  cos(gamma), 0]
[         0,           0, 1]

>> RyBeta = [ cos(beta) 0 sin(beta); 0 1 0; -sin(beta) 0 cos(beta)]

RyBeta =

[ cos(beta), 0, sin(beta)]
[         0, 1,         0]
[-sin(beta), 0, cos(beta)]

>> RzAlpha = [ cos(alpha) -sin(alpha) 0; sin(alpha) cos(alpha) 0; 0 0 1]

RzAlpha =

[cos(alpha), -sin(alpha), 0]
[sin(alpha),  cos(alpha), 0]
[         0,           0, 1]

>> RyBeta*RzAlpha

ans =

[ cos(alpha)*cos(beta), -cos(beta)*sin(alpha), sin(beta)]
[           sin(alpha),            cos(alpha),         0]
[-cos(alpha)*sin(beta),  sin(alpha)*sin(beta), cos(beta)]

>> RzGamma*RyBeta*RzAlpha

ans =

[cos(alpha)*cos(beta)*cos(gamma) - sin(alpha)*sin(gamma), - cos(alpha)*sin(gamma) - cos(beta)*cos(gamma)*sin(
alpha), cos(gamma)*sin(beta)]
[cos(gamma)*sin(alpha) + cos(alpha)*cos(beta)*sin(gamma),   cos(alpha)*cos(gamma) - cos(beta)*sin(alpha)*sin(g
amma), sin(beta)*sin(gamma)]
[                                  -cos(alpha)*sin(beta),                                  sin(alpha)*sin(beta),            cos(beta)]

>>
>>
>>
>>
>> RzGamma_ = [ cos(gamma - pi) -sin(gamma - pi ) 0; sin(gamma -pi ) cos(gamma - pi) 0; 0 0 1]

RzGamma_ =

[-cos(gamma),  sin(gamma), 0]
[-sin(gamma), -cos(gamma), 0]
[          0,           0, 1]

>> RyBeta_ = [ cos(-beta) 0 sin(-beta); 0 1 0; -sin(-beta) 0 cos(-beta)]
```

```
RyBeta_ =

[cos(beta), 0, -sin(beta)]
[        0, 1,          0]
[sin(beta), 0,  cos(beta)]

>> RzAlpha_ = [ cos(alpha -pi ) -sin(alpha - pi) 0; sin(alpha - pi) cos(alpha - pi) 0; 0 0 1]

RzAlpha_ =

[-cos(alpha),  sin(alpha), 0]
[-sin(alpha), -cos(alpha), 0]
[          0,           0, 1]

>> RyBeta_*RzAlpha_

ans =

[-cos(alpha)*cos(beta), cos(beta)*sin(alpha), -sin(beta)]
[          -sin(alpha),          -cos(alpha),          0]
[-cos(alpha)*sin(beta), sin(alpha)*sin(beta),  cos(beta)]

>>
```