Q2

The functions are

$$f_1 = a\cos(\theta) + b\cos(\theta+\phi) - x_p$$
$$f_2 = a\sin(\theta) + b\sin(\theta+\phi) - y_p$$

And their partial derivatives are

$$\frac{\partial f_1}{\partial \theta} = -a\sin(\theta) - b\sin(\theta+\phi)$$
$$\frac{\partial f_1}{\partial \phi} = -b\sin(\theta+\phi)$$
$$\frac{\partial f_2}{\partial \theta} = a\cos(\theta) + b\cos(\theta+\phi)$$
$$\frac{\partial f_2}{\partial \phi} = b\cos(\theta+\phi)$$

So the Jacobian matrix will be

$$\begin{bmatrix} \frac{\partial f_1}{\partial \theta} & \frac{\partial f_1}{\partial \phi} \\ \frac{\partial f_2}{\partial \theta} & \frac{\partial f_2}{\partial \phi} \end{bmatrix} \iff \begin{bmatrix} -a\sin(\theta) - b\sin(\theta+\phi) & -b\sin(\theta+\phi) \\ a\cos(\theta) + b\cos(\theta+\phi) & b\cos(\theta+\phi) \end{bmatrix}$$

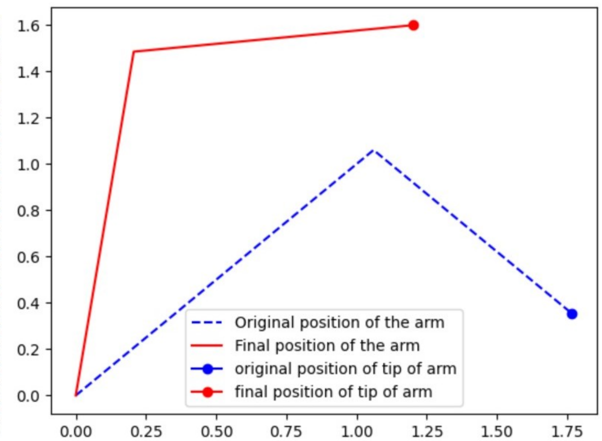
The code below is written in Python

```
import pandas as pd
import numpy as np
import scipy.linalg as sci
import matplotlib.pyplot as plt
from math import sin, cos, pi, sqrt

def robotarm( a, b, p, r, maxit, tol):
    guess_v = r
    x_p, y_p = p[0], p[1]

    f_1 = lambda theta, phi, a, b, x_p: a*cos(theta) + b*cos(theta + phi) - x_p
    f_2 = lambda theta, phi, a, b, y_p: a*sin(theta) + b*sin(theta + phi) - y_p

    def f(v:list, a:float, b:float, x_p:float, y_p:float, f_1=f_1, f_2=f_2) -> list:
        # v is a matrix
        theta, phi = v[0], v[1]
        return np.array([f_1(theta, phi, a, b, x_p), f_2(theta, phi, a, b, y_p)])


    # Partial Derivative functions

    d_f1_theta = lambda theta, phi, a, b: -a*sin(theta) - b*sin(theta + phi)
    d_f1_phi = lambda theta, phi, b: -b*sin(theta + phi)

    d_f2_theta = lambda theta, phi, a, b: a*cos(theta) + b*cos(theta + phi)
    d_f2_phi = lambda theta, phi, b: b*cos(theta + phi)

    def Jac(v:list, a:float, b: float, d_f1_t = d_f1_theta,
            d_f1_p=d_f1_phi, d_f2_t=d_f2_theta, d_f2_p=d_f2_phi):

        theta, phi = v[0], v[1]
        row_1 = [ d_f1_t(theta, phi, a, b), d_f1_p(theta, phi, b) ]
        row_2 = [ d_f2_t(theta, phi, a, b), d_f2_p(theta, phi, b) ]

        return np.array([row_1, row_2])


    def delta_solver(original_v: list, a:float, b:float, x_p:float,
                     y_p:float, J=Jac, f=f):

        f_vect = f(original_v, a, b, x_p, y_p)
        Jac_mat = J(original_v, a, b)

        delta_v = sci.solve(Jac_mat, -1*f_vect)
        new_v = delta_v + original_v

        return new_v

    v = guess_v
    count = 0
    while count < maxit and np.linalg.norm(f(v, a, b, x_p, y_p)) > tol:
        v = delta_solver(v, a, b, x_p, y_p)
        count += 1
```

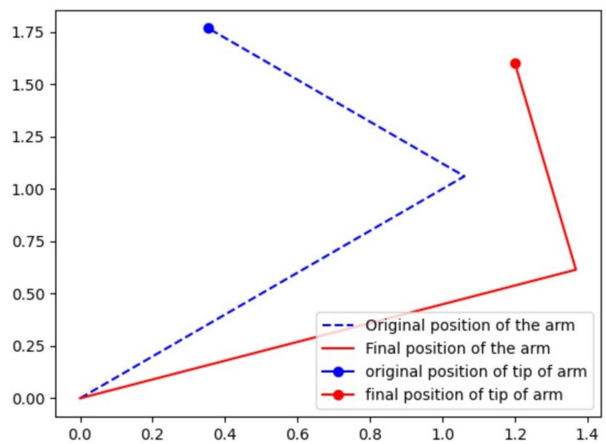
```
    return v


def Output(a, b, p, r, maxit = 20, tol = 10**(-8)):

    data = []
    f_1 = lambda theta, phi, a, b, x_p: a*cos(theta) + b*cos(theta + phi) - x_p
    f_2 = lambda theta, phi, a, b, y_p: a*sin(theta) + b*sin(theta + phi) - y_p

    def f(v:list, a:float, b:float, x_p:float, y_p:float, f_1=f_1, f_2=f_2) -> list:
        # v is a matrix
        theta, phi = v[0], v[1]
        return np.array([f_1(theta, phi, a, b, x_p), f_2(theta, phi, a, b, y_p)])

    for i in range(maxit+1):
        v = robotarm(a, b, p, r, i, tol)
        theta, phi = v[0], v[1]
        data.append([theta, phi, np.linalg.norm(f(v, a, b, p[0], p[1]))/np.linalg.norm(
                    f(r, a, b, p[0], p[1])), a*cos(theta) + b*cos(theta + phi),
                    a*sin(theta) + b*sin(theta + phi)])

    or_x = [0, a*cos(data[0][0]), data[0][3]] # original vector data
    or_y = [0, a*sin(data[0][0]), data[0][4]]

    x = [0, a*cos(data[-1][0]), data[-1][3]]  # final vector data
    y = [0, a*sin(data[-1][0]), data[-1][4]]

    plt.plot(or_x, or_y, linestyle='dashed', color='blue')
    plt.plot(x, y, linestyle='solid', color='red')
    plt.plot(data[0][3], data[0][4], color='blue', marker='o')
    plt.plot(data[-1][3], data[-1][4], color='red', marker='o')
    plt.legend(['Original position of the arm', 'Final position of the arm',
               'original position of tip of arm', 'final position of tip of arm'])
    plt.show()

    df = pd.DataFrame(data, columns = ['(theta)', '(phi)',
                    'residual ||f(x^(k))||/||f(x^(0))||', 'x-coord', 'y-coord'])
    if data[-1][2]>tol:
        out_row = 0
        min  = data[0][2]
        for row in data:
            if row[2]<min:
                min=row[2]
                out_row = row
        out_row[0] = round(out_row[0], 6)
        out_row[1] = round(out_row[1], 6)
        out_row[2] = round(out_row[2], 6)
        out_row[3] = round(out_row[3], 6)
        out_row[4] = round(out_row[4], 6)

        string = f'{out_row}'[1:-1]
        # The Warning of non-convergence
        print(f'WARNING: The sequence did not converge within the tolerance, \
```

```
        \n however the most accurate result (according to the residual) is \n {string}')
    return df

r = np.array([pi/4, -pi/2])
r_other = np.array([pi/4, pi/2])
a, b = 1.5, 1

p = np.array([1.2, 1.6])
print(Output(a, b, p, r))
print()

p = np.array([1.2, 1.6])
print(Output(a, b, p, r_other))
print()

t = sqrt((a+b)**2-1.2**2)
p = np.array([1.2, t])
print(Output(a, b, p, r))
print()

p = np.array([1.2, 2.2])
print(Output(a, b, p, r))
```

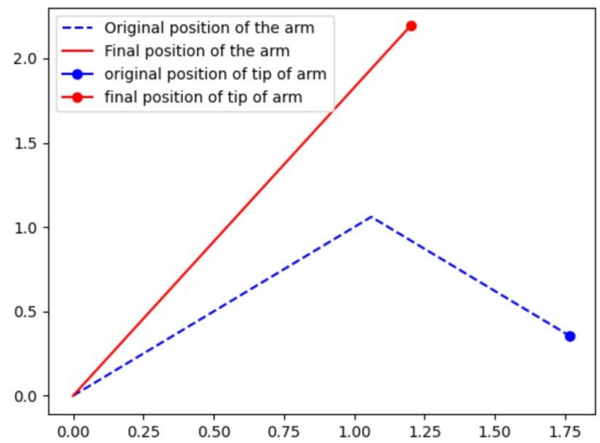
	θ (theta)	ϕ (phi)	residual ||f(x^(k))||/||f(x^(0))||	x-coord	y-coord
0	0.785398	-1.570796	1.000000e+00	1.767767	0.353553
1	1.640627	-1.946126	3.909536e-01	0.849037	1.195575
2	1.487949	-1.386143	5.922957e-02	1.118951	1.596485
3	1.433024	-1.320832	1.660260e-03	1.199718	1.597744
4	1.432658	-1.318119	2.029931e-06	1.199997	1.600000
5	1.432656	-1.318116	2.077518e-12	1.200000	1.600000
6	1.432656	-1.318116	2.077518e-12	1.200000	1.600000
7	1.432656	-1.318116	2.077518e-12	1.200000	1.600000
8	1.432656	-1.318116	2.077518e-12	1.200000	1.600000
9	1.432656	-1.318116	2.077518e-12	1.200000	1.600000
10	1.432656	-1.318116	2.077518e-12	1.200000	1.600000
11	1.432656	-1.318116	2.077518e-12	1.200000	1.600000
12	1.432656	-1.318116	2.077518e-12	1.200000	1.600000
13	1.432656	-1.318116	2.077518e-12	1.200000	1.600000
14	1.432656	-1.318116	2.077518e-12	1.200000	1.600000
15	1.432656	-1.318116	2.077518e-12	1.200000	1.600000
16	1.432656	-1.318116	2.077518e-12	1.200000	1.600000
17	1.432656	-1.318116	2.077518e-12	1.200000	1.600000
18	1.432656	-1.318116	2.077518e-12	1.200000	1.600000
19	1.432656	-1.318116	2.077518e-12	1.200000	1.600000
20	1.432656	-1.318116	2.077518e-12	1.200000	1.600000



	θ (theta)	ϕ (phi)	residual ||f(x^(k))||/||f(x^(0))||	x-coord	y-coord
0	0.785398	1.570796	1.000000e+00	0.353553	1.767767
1	0.307293	1.569002	2.378766e-01	1.128964	1.407417
2	0.415494	1.334841	1.424174e-02	1.193800	1.589389
3	0.421893	1.318202	7.740364e-05	1.199981	1.599936
4	0.421935	1.318116	2.103286e-09	1.200000	1.600000
5	0.421935	1.318116	2.103286e-09	1.200000	1.600000
6	0.421935	1.318116	2.103286e-09	1.200000	1.600000
7	0.421935	1.318116	2.103286e-09	1.200000	1.600000
8	0.421935	1.318116	2.103286e-09	1.200000	1.600000
9	0.421935	1.318116	2.103286e-09	1.200000	1.600000
10	0.421935	1.318116	2.103286e-09	1.200000	1.600000
11	0.421935	1.318116	2.103286e-09	1.200000	1.600000
12	0.421935	1.318116	2.103286e-09	1.200000	1.600000
13	0.421935	1.318116	2.103286e-09	1.200000	1.600000
14	0.421935	1.318116	2.103286e-09	1.200000	1.600000
15	0.421935	1.318116	2.103286e-09	1.200000	1.600000
16	0.421935	1.318116	2.103286e-09	1.200000	1.600000
17	0.421935	1.318116	2.103286e-09	1.200000	1.600000
18	0.421935	1.318116	2.103286e-09	1.200000	1.600000
19	0.421935	1.318116	2.103286e-09	1.200000	1.600000
20	0.421935	1.318116	2.103286e-09	1.200000	1.600000

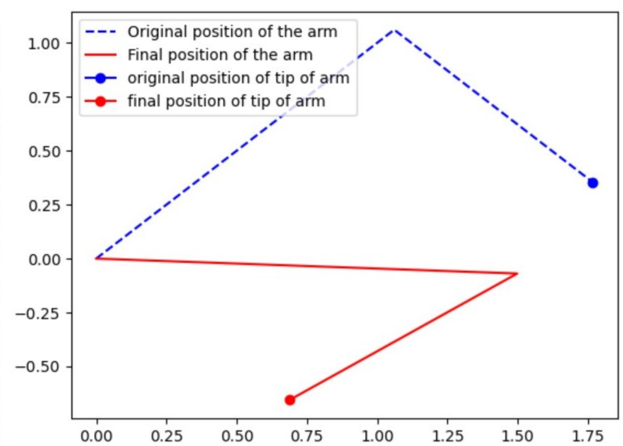


	θ (theta)	ϕ (phi)	residual ||f(x^(k))||/||f(x^(0))||	x-coord	y-coord
0	0.785398	-1.570796	1.000000e+00	1.767767	0.353553
1	1.920250	-1.806314	5.109257e-01	0.479939	1.523030
2	1.377548	-0.869635	1.235279e-01	1.161833	1.958433
3	1.248842	-0.429844	3.020478e-02	1.157584	2.153391
4	1.156014	-0.214255	7.152827e-03	1.192852	2.181401
5	1.112986	-0.107013	1.785108e-03	1.198245	2.190216
6	1.091543	-0.053492	4.459220e-04	1.199575	2.192425
7	1.080840	-0.026744	1.114582e-04	1.199895	2.192984
8	1.075490	-0.013372	2.786313e-05	1.199974	2.193124
9	1.072816	-0.006686	6.965695e-06	1.199994	2.193159
10	1.071479	-0.003343	1.741418e-06	1.199998	2.193168
11	1.070810	-0.001671	4.353542e-07	1.200000	2.193170
12	1.070476	-0.000836	1.088385e-07	1.200000	2.193171
13	1.070309	-0.000418	2.720963e-08	1.200000	2.193171
14	1.070225	-0.000209	6.802407e-09	1.200000	2.193171
15	1.070183	-0.000104	1.700602e-09	1.200000	2.193171
16	1.070183	-0.000104	1.700602e-09	1.200000	2.193171
17	1.070183	-0.000104	1.700602e-09	1.200000	2.193171
18	1.070183	-0.000104	1.700602e-09	1.200000	2.193171
19	1.070183	-0.000104	1.700602e-09	1.200000	2.193171
20	1.070183	-0.000104	1.700602e-09	1.200000	2.193171



WARNING: The sequence did not converge within the tolerance,
however the most accurate result (according to the residual) is
1.070873, 0.001553, 0.003103, 1.197032, 2.194792

	θ (theta)	ϕ (phi)	residual ||f(x^(k))||/||f(x^(0))||	x-coord	y-coord
0	0.785398	-1.570796	1.000000	1.767767	0.353553
1	1.923469	-1.804977	0.512433	0.474844	1.526165
2	1.375283	-0.863939	0.125273	1.163493	1.960772
3	1.244736	-0.415453	0.031387	1.155876	2.158414
4	1.144854	-0.182946	0.008304	1.191725	2.186259
5	1.086205	-0.036721	0.003313	1.196788	2.194465
6	0.969997	0.253653	0.013080	1.188168	2.177674
7	1.036466	0.087140	0.004285	1.196330	2.192581
8	1.099866	-0.071098	0.003887	1.196449	2.193384
9	1.029480	0.104971	0.004813	1.195527	2.191849
10	1.088511	-0.042727	0.003386	1.196932	2.194223
11	0.986492	0.212421	0.010103	1.190800	2.182789
12	1.047768	0.058979	0.003644	1.196831	2.193713
13	1.127390	-0.139889	0.006140	1.194305	2.189596
14	1.070873	0.001553	0.003103	1.197032	2.194792
15	3.643985	-6.431340	2.461344	-2.252561	-1.069161
16	23.390078	-57.340673	2.495350	-1.077939	-2.048245
17	27.412086	-67.790711	1.770901	-1.871226	0.693151
18	25.060413	-64.558031	1.699655	1.270331	-1.082583
19	27.207594	-67.016324	1.551812	-1.237597	0.455131
20	25.086538	-65.301057	1.501660	0.688130	-0.655339



Here we can see that for the first three iterations of the code, the code approximates the solution extremely well. For the first two iterations of the code, the sequence of answers converges over the tolerance threshold within the first few iterations while the second one takes 13 iterations to reach it. On the other hand, the last iteration doesn't converge at all. The errors for the first 3 iterations decrease exponentially while the error of the last iteration decreases exponentially and later just hovers at about the same level of 10^{-3}. The interesting part is that as it after it reaches the 14th iteration, the accuracy decreases rapidly to approximately 2.5 and the x, y coordinates of the convergence become about -2.25 and -1.06 while as the actual coordinates are 1.2 and 2.2.

The reason to why the there is this sudden decrease in accuracy is that the Newton's method is extremely input sensitive. We can look at the Jacobian matrices of the 14th and 15th iteration ans see that the entries are at most 10^{-2} away from each other.

$$\begin{bmatrix} -2.1896 & -0.8347 \\ 1.1943 & 0.5501 \end{bmatrix} ; \begin{bmatrix} -2.1948 & -0.8784 \\ 1.1943 & 0.4800 \end{bmatrix}$$

While on the other hand, the vectors of functions themselves are wildly different from one another

$$\begin{bmatrix} -1.006 \\ 0.986 \end{bmatrix} \approx \begin{bmatrix} -1 \\ 1 \end{bmatrix} ; \begin{bmatrix} -0.003 \\ -1.044 \end{bmatrix} \approx \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

Thus having two matrices that are approximately the same augmented with two completely different vectors results in having two completely different delta vector and considering that the values of both 13th and 14th