HW10

April 19, 2022

1 Assignment 10

Daniel González Esparza

Zainab Alsuwaykit

2 Q1: Smallest Enclosing Ellipse.

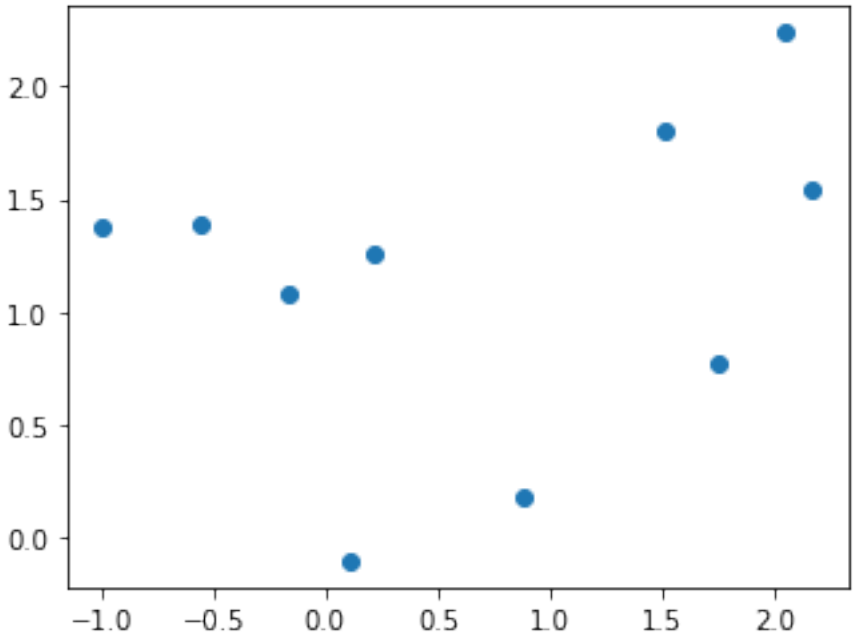
$$\begin{array}{ll} \text{minimize} & -\log\det A \\ \textit{subject to} & \|Ax_i + b\|_2 \leq 1 \\ & i = 1, \ldots, m \end{array}$$

\

$$\begin{array}{ll} \text{minimize} & -\log\det A \\ \textit{subject to} & \|A(data[i] - c)\|_2 \leq 1 \\ & i = 1, \ldots, m \\ & c = -A^{-1}b \end{array}$$

[2]:
```
#Q1

import numpy as np
import matplotlib
import matplotlib.pyplot as plt

np.random.seed(421)            # seed the random number generator, for␣
 ↪repeatability
m = 10
n = 2

# generate random points in a 4x2 box centered at the origin
data =  np.random.rand(m, 2) * np.array([4, 2]) - [2, 1]
# rotate by pi/10 and translate the points
ang = np.pi/10
R = np.array([ [np.cos(ang), np.sin(ang)], [-np.sin(ang), np.cos(ang)] ])
data = data @ R + [1, 1]

# plot the data points
plt.scatter(data[:,0], data[:,1])
```

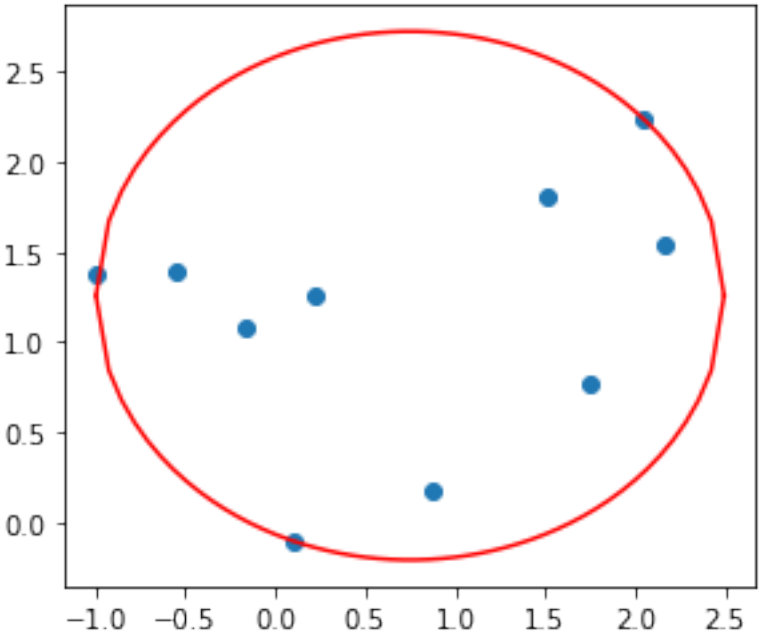
```
ax = plt.gca()
ax.set_aspect('equal')
plt.show()


# Formulation and solution:
# Your work goes here
```



[3]:
```
import cvxpy as cp
from numpy import linalg as la
x = cp.Variable((2,1))
A = cp.diag(x)
b = cp.Variable((2,1))
c = cp.Variable((2,1))
M=np.identity(2)

equation = -cp.log_det(A)
constraints = []
for i in range(0,len(data)):
    constraints.append(cp.quad_form((A@np.reshape(data[i],(2,1))-b),M)<=1)

prob = cp.Problem(cp.Minimize(equation),constraints)
sol = prob.solve()

a=1/x.value
c=b.value/x.value
```

```
print(c)
a1=a[0]
a2=a[1]
h=c[0]
k=c[1]
xx=np.linspace(h-a1,h+a1,50)
yy=a[1]*np.sqrt(1-((xx-c[0])/a[0])**2)+c[1]
yy2=-a[1]*np.sqrt(1-((xx-c[0])/a[0])**2)+c[1]
plt.plot(xx,yy,'r-', label='fitted curve')
plt.plot(xx,yy2,'r-')
plt.scatter(data[:,0], data[:,1])
ax = plt.gca()
ax.set_aspect('equal')
plt.show()
```

```
[[0.74962467]
 [1.25673266]]
```



• Can you use the Lagrange multipliers to recognize which points lie on the boundary of the ellipse? Comment.

[4]:
```
for i in range(0,len(data)):
    print("constraint ", i , " : ",constraints[i].dual_value)
```

```
constraint  0  :  [2.08210883e-07]
constraint  1  :  [0.20807884]
```

```
constraint  2  :  [0.34275195]
constraint  3  :  [0.]
constraint  4  :  [3.08469465e-07]
constraint  5  :  [0.]
constraint  6  :  [0.449159]
constraint  7  :  [2.27808268e-07]
constraint  8  :  [0.]
constraint  9  :  [2.87562799e-07]
```

We have only three values of the lagrange multipliers that are not zero which means they are active. Therefore, three points lie on the boundary of the ellipse.

---

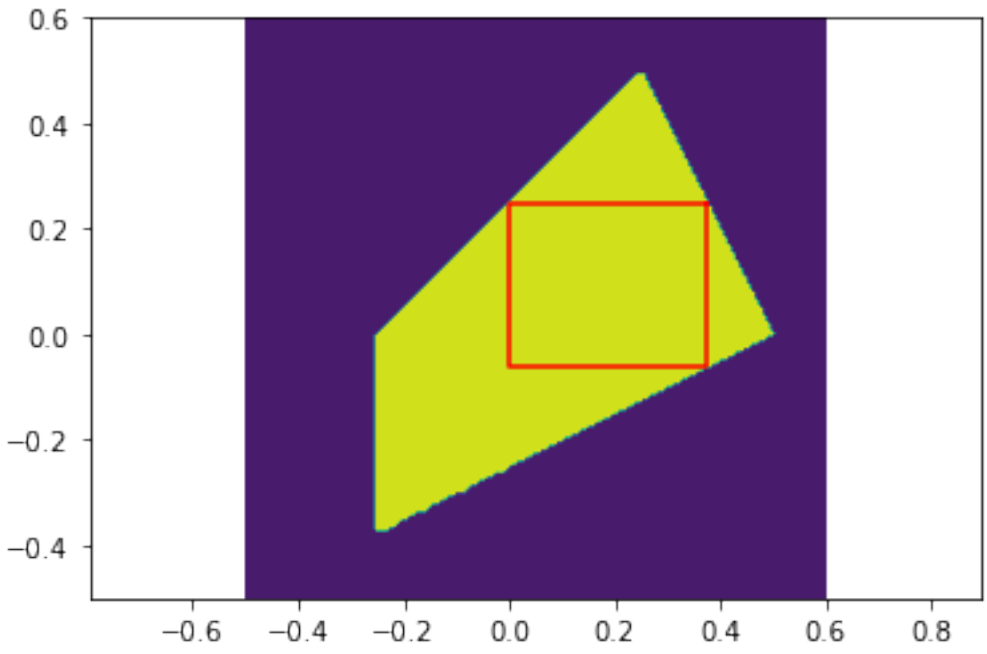
3 Q2: Largest Enclosed Rectangle.

• Formulate an inequality constrained optimization problem in terms of the coordinates of the lower- left and upper right corners (or something equivalent). Use an appropriate routine or your code above to solve the problem for the following polygon Ax b, where:

[6]:
```
#Q2
A = np.array([[0,-1],[2,-4],[2,1],[-4,4],[-4,0]] )
b = 1
lw = cp.Variable((2,1))
corner = cp.Variable((2,1))
mat2 = [[1,0],[0,0]]
mat4 = [[0,0],[0,1]]
corner2 = cp.vstack(corner-mat2@lw)
corner3 = cp.vstack(corner-lw)
corner4 = cp.vstack(corner-mat4@lw)







B = cp.diag(lw)

equation = -cp.log_det(B)
constraints = [A@corner -b <=0,A@corner3-b<=0, A@corner2-b<=0,A@corner4-b<=0]

prob = cp.Problem(cp.Minimize(equation),constraints)
sol = prob.solve()
print("the solution: ",sol)



x,y=np.meshgrid(np.linspace(-0.5,0.6,150),np.linspace(-0.5,0.6,150))

bo1 = np.zeros(np.shape(y))
```

```
s=(A[0,0]*x+A[0,1]*y<b)\
&(A[1,0]*x+A[1,1]*y<b)\
&(A[2,0]*x+A[2,1]*y<b)\
&(A[3,0]*x+A[3,1]*y<b)\
&(A[4,0]*x+A[4,1]*y<b)
bo1[s] = 1




plt.figure()
plt.contourf(x,y,bo1)
plt.plot([corner.value[0], corner2.value[0],corner3.value[0],corner4.
 ↪value[0],corner.value[0]],[corner.value[1], corner2.value[1],corner3.
 ↪value[1],corner4.value[1],corner.value[1]],'r')
plt.axis('equal')
plt.show()
```

```
the solution:  2.143998738120301
```



- As formulated above, the problem involves an exponential number of constraints in the number of spatial dimensions (the number of the vertices of the Rd rectangle grows as 2d). Formulate the problem in a way that avoids this combinatorial explosion.

Consider the upper and lower corner as u_i and l_i respectivly. The problem can be written as:

$$\begin{array}{ll} \text{maximize} & \sum_{i=1}^{2} \log(u_i - l_i) \\ \textit{subject to} & Av_k - b \leq 0 \\ & i = 1, \ldots, m \end{array}$$

that is :

$$\begin{array}{ll} \text{minimize} & -\sum_{i=1}^{2} \log(u_i - l_i) \\ \textit{subject to} & Av_k - b \leq 0 \\ & i = 1, \ldots, m \end{array}$$

Looking at the constraints, we can write the matrix A as

$$A = A^{+} - A^{-}$$

-if the element of A is positive, the largest value of v is u, -if the element of A is negative, the largest value of v is l. So each time I have to check if the element a form A is positive, make it v=u, otherwise, v=l the problem can be then written as:

$$\begin{array}{ll} \text{minimize} & -\sum_{i=1}^{2} \log(u_i - l_i) \\ \textit{subject to} & (A^{+}u - A^{-}l) - b \leq 0 \end{array}$$

4 Q3: Planar Disk Contact

• Determine the appropriate variables and constraints and formulate the problem as an appropriate optimization problem with a quadratic objective and quadratic constraints. Is the problem convex?

$$\begin{array}{lll} \text{minimize} & & \sum_{j=1}^{9} \frac{1}{2} k_j \Delta_j^2 \\ \text{subject to} & \Delta \geq b & \\ & & \\ \text{whare} & \begin{bmatrix} \sqrt{2}r1 \\ r1 + r4 \\ \sqrt{2}r4 \\ 2r1 \\ r1 + r4 \\ r1 + r4 \\ \sqrt{2}r1 \\ 2r1 \\ \sqrt{2}r1 \end{bmatrix} & \text{and} \quad \Delta = \begin{bmatrix} \|x1 - q1\| \\ \|x4 - x1\| \\ \|x4 - q4\| \\ \|x1 - x2\| \\ \|x4 - x2\| \\ \|x4 - x3\| \\ \|x3 - q3\| \\ \|x3 - x2\| \\ \|x2 - q2\| \end{bmatrix} \end{array}$$

This problem is not convex.

• Write the Lagrangian of the problem. What is the significance of the multipliers?

The lagrangian :

$$\frac{1}{2}\Delta^t K \Delta + \lambda^t (\Delta^2 - b^2)$$

Considering the multipliers, we can determine whether the disks are in contact or not.

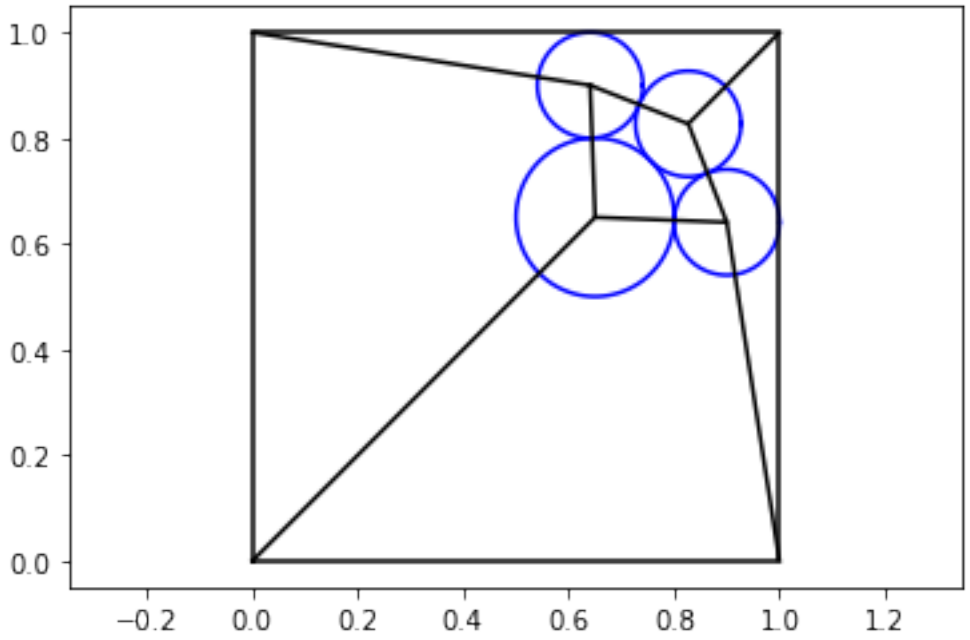
[2]:
```
import numpy as np
import matplotlib.pyplot as plt
from numpy import linalg as la
import scipy
```

```
import scipy.optimize as optimize

def Potential(unknowns):
    x1, y1, x2, y2, x3, y3, x4, y4= unknowns
    fo= x1**2 + (1-y1)**2
    fo+= (x4-x1)**2 + (y4-y1)**2
    fo+= (x4-x3)**2 + (y4-y3)**2
    fo+= (x4)**2 + (y4)**2
    fo+= (1-x3)**2 + (y3)**2
    fo+= 10*((x2-x1)**2 + (y2-y1)**2)
    fo+= 10*((1-x2)**2 + (1-y2)**2)
    fo+= 10*((x3-x2)**2 + (y3-y2)**2)
    fo+= 10*((x4-x2)**2 + (y4-y2)**2)
    return 0.5*fo
#Conditions-------------------------------------------------------------------------------
q1=[0,1]
q2=[1,1]
q3=[1,0]
q4=[0,0]
r1=0.1
r4=0.15
b=np.reshape([np.sqrt(2)*r1,r1+r4,np.sqrt(2)*r4,2*r1,r1+r4,r1+r4,np.
 ↪sqrt(2)*r1,2*r1,np.sqrt(2)*r1],(9,1))
b=b**2
#Constraints-----------------------------------------------------------------------------

cons = ({'type': 'ineq', 'fun': lambda x: (x[0]-q1[0])**2 + (x[1]-q1[1])**2 ␣
 ↪-b[0]},
        {'type': 'ineq', 'fun': lambda x: (x[6]-x[0])**2  + (x[7]-x[1])**2  ␣
 ↪-b[1]},
        {'type': 'ineq', 'fun': lambda x: (x[6]-q4[0])**2 + (x[7]--q4[1])**2␣
 ↪-b[2]},
        {'type': 'ineq', 'fun': lambda x: (x[0]-x[2])**2  + (x[1]-x[3])**2  ␣
 ↪-b[3]},
        {'type': 'ineq', 'fun': lambda x: (x[6]-x[2])**2  + (x[7]-x[3])**2  ␣
 ↪-b[4]},
        {'type': 'ineq', 'fun': lambda x: (x[6]-x[4])**2  + (x[7]-x[5])**2  ␣
 ↪-b[5]},
        {'type': 'ineq', 'fun': lambda x: (x[4]-q3[0])**2 + (x[4]-q3[1])**2 -␣
 ↪b[6]},
        {'type': 'ineq', 'fun': lambda x: (x[4]-x[2])**2  + (x[5]-x[3])**2 -␣
 ↪b[7]},
        {'type': 'ineq', 'fun': lambda x: (x[2]-q2[0])**2 + (x[3]-q2[1])**2 -␣
 ↪b[8]})
```

```
bnds = ((r1,1-r1),(r1,1-r1),(r1, 1-r1),(r1, 1-r1),(r1, 1-r1),(r1, 1-r1),(r4,␣
↪1-r4),(r4, 1-r4))

#Solution-------------------------------------------------------------------------------

result = optimize.minimize(Potential, [0,1,1,1,1,5,1,1], method='trust-constr',␣
↪constraints=cons, bounds=bnds)

if result.success:
    fitted_params = result.x
    x1, y1, x2, y2, x3, y3, x4, y4 = fitted_params
else:
    raise ValueError(result.message)

print(x1, y1, x2, y2, x3, y3, x4, y4)

#Plots-----------------------------------------------------------------------------
theta = np.linspace(0, 2 * np.pi, 100)
radius = r1
xx = radius * np.cos(theta)+x1
yy = radius * np.sin(theta)+y1
plt.plot(xx, yy,'b')
plt.plot([0,x1],[1,y1],'k')

radius = r1
xx = radius * np.cos(theta)+x2
yy = radius * np.sin(theta)+y2
plt.plot(xx, yy,'b')
plt.plot([1,x2],[1,y2],'k')

radius = r1
xx = radius * np.cos(theta)+x3
yy = radius * np.sin(theta)+y3
plt.plot(xx, yy,'b')
plt.plot([1,x3],[0,y3],'k')

radius = r4
xx = radius * np.cos(theta)+x4
yy = radius * np.sin(theta)+y4
plt.plot(xx, yy,'b')
plt.plot([0,x4],[0,y4],'k')

plt.plot([0,1,1,0,0],[0,0,1,1,0],'k')
plt.plot([x1,x2,x3,x4,x1],[y1,y2,y3,y4,y1],'k')

plt.axis('equal')
plt.show()
```

```
C:\Users\dgonz\anaconda3\lib\site-
packages\scipy\optimize\_hessian_update_strategy.py:182: UserWarning: delta_grad
== 0.0. Check if the approximated function is linear. If the function is linear
better results can be obtained by defining the Hessian as zero instead of using
quasi-Newton approximations.
  warn('delta_grad == 0.0. Check if the approximated '
```

```
0.640770131188371 0.8999999257289535 0.8269535673951677 0.8269537262888497
0.8999998749712528 0.6407703209332815 0.6501768524564332 0.6501769363076084
```



We set up the objective and the constraints, however, the constraint where the distances between the radius is limited. The cvxpy does not accept the formualtion of this constraint.

---

5 Q4: Trust Region Constraint.

$$\begin{array}{ll} \textit{minimize} & p^t H p + 2 g^t p \\ \text{subject to} & p^t p \leq \Delta^2 \end{array}$$

derive the dual problem

$$\begin{array}{ll} \text{maximize} & -g^t (H + \lambda I)^{-1} g - \lambda \Delta^2 \\ \text{subject to} & H + \lambda I \succeq 0 \end{array}$$

• Write the Lagrangian and derive the dual problem note: this problem is not convex when H is not positive definite The langrangian:

$$L(p, \lambda) = p^T H p + 2 g^T p + \lambda \left(p^T p - \Delta^2 \right) \qquad = p^T (H + \lambda I) p + 2 g^T p - \lambda \Delta^2$$

The gradient of the langrangian:

$$\nabla L(p) = 2(H + \lambda I)p + 2g \therefore p = -(H + \lambda I)^{-1}g$$

substitute in the langrangian, the dual problem:

$$g(\lambda) = g^T(H+\lambda I)^{-1}(H+\lambda I)(H+\lambda I)^{-1}g - 2g^T(H+\lambda I)^{-1}g - \lambda\Delta^2 \quad = g^T(H+\lambda I)^{-1}g - 2g^T(H+\lambda I)^{-1}g - \lambda\Delta^2 \quad = -$$

- Express this dual in the computationally oriented form:

We start from the the constraints, For H to be positive definite, the smallest eigen value of H has to be positive. If we have H positive definite, no restriction on , however, otherwise > smallest eigen value of H. so we can consider this as our constraint which is linear. (finding the eigen values is linear) In addition, in the objective function, inv(H) can be replced by its eigen vectors and value which is basicaly the reciprocal of the eigen value of H.

Since H is symmetric and positive definite, we can write it as:

$$\begin{array}{l}(H + \lambda I) = q\lambda q^{-1} \\ (H + \lambda I)^{-1} = q^{-1}\lambda_i^{-1}q \\ \therefore g(\lambda) = -g^t q^{-1}(\lambda_i + \lambda)^{-1}qg - \lambda\Delta^2 \\ \text{Since } q^{-1} = q^t, then: \\ g(\lambda) = \frac{-g^t q^t qg}{\lambda_i+\lambda} - \lambda\Delta^2\end{array}$$

$$\begin{array}{ll}\text{maximize} & \sum_{i=1}^{n} \frac{-\left(q_i^t g\right)^2}{(\lambda_i+\lambda)} - \lambda\Delta^2 \\ \text{subject to} & \lambda \geq -\lambda_{\min}(H)\end{array}$$

- Describe in pseudo-code how to solve this problem and find the optimal minimizer p.

1- choose a starting point x^0, and p^k, calculate the hessian and gradient at x^k

2- while $\lambda \geq -\lambda_{\min}(H)$

3- calculate p^k = maximize: $\sum_{i=1}^{n} \frac{-\left(q_i^t g\right)^2}{(\lambda_i+\lambda)} - \lambda\Delta^2$

4- check the agreement between the objective f(x) and the quadratic model m_k(p) $\rho = \frac{f\left(x^k\right) - f\left(x^k+p^k\right)}{m_k(0) - m_k\left(p^k\right)}$

if $\rho \leq 0.25$ and $\Delta^k + 1$ is reduced then , $x^{k+1} = x^k$

else if $\rho \geq 0.75$ and $\Delta^k + 1$ is enlarged then,$x^{k+1} = x^k + p^k$

otherwise keep same trust region and $x^{k+1} = x^k + p^k$

---

6 Q5: Levenberg Marquardt.

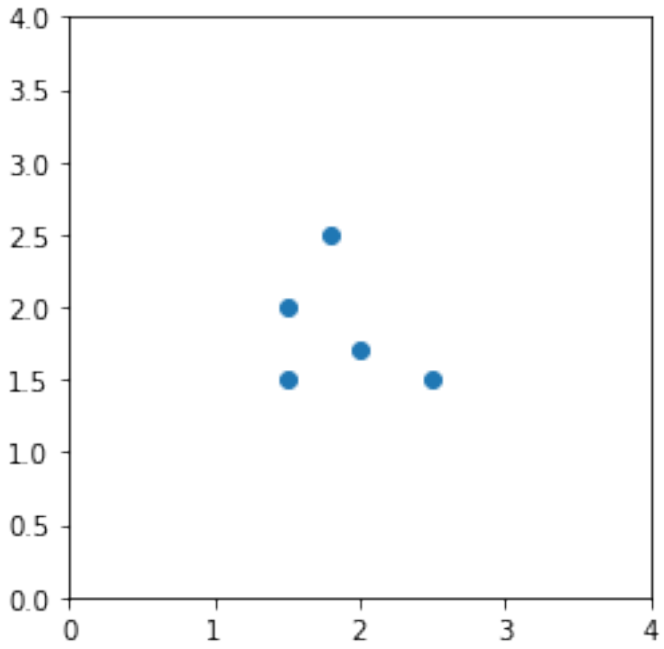
- Code for a Levenberg Marquardt routine is posted on on Blackboard. Describe in a few words the k update strategy used in it, and what happens at every iteration k after solving (1).

[4]:
```
import numpy as np

# Takes two routines (f, Df) that return the residual vector and its Jacobian,
# a starting point (x0), and an initial value for the trust parameter (lambda0)
def levenberg_marquardt(f,r, Df, x0, lambda0, kmax=100, tol=1e-6):
    n = len(x0)
    x = x0
    lam = lambda0
    l=[lam]
    obj = np.zeros((0,1))
    res = np.zeros((0,1))
    for k in range(kmax):
        obj = np.vstack([obj, np.linalg.norm(r(x))**2])
        res = np.vstack([res, np.linalg.norm(2*Df(x).T @ r(x))])
        if np.linalg.norm(2*Df(x).T @ r(x)) < tol:
            break
        xt = x - np.linalg.inv((Df(x).T @ Df(x)) + lam*np.eye(n)) @ (Df(x).T @␣
↪r(x))
        if np.linalg.norm(r(xt)) < np.linalg.norm(r(x)):
            lam = 0.8 * lam
            x = xt
            l=np.vstack((l,[lam]))
        else:
            lam = 2.0 * lam
            l=np.vstack((l,[lam]))
    return x, obj, res,l
```

The Levenberg Marquardt method uses the general idea of adding a trust region constraints at every iteration of the original Gauss Newton method. When we far away from the solution, we move to the gradient direction, and when we get closer and closer to the solution, I use Gauss Newton. This is done by adding a panalty term to the hessian that works as regularization of the hessain when it is not well conditoned. We decrease or increase the panalty term based on how far we are from optimality. The strategy used in this code is we take gradient step when is large, that is by decreasing my step by Gauss-Newton solution at the current iteration. Otherwise, no change in my current step.

[385]:
```
import numpy as np
import matplotlib.pyplot as plt

# five locations yi in a 5 by 2 matrix
A = np.array([[1.8, 2.5], [2.0, 1.7], [1.5, 1.5], [1.5, 2.0], [2.5, 1.5]])
# vector of measured distances to five locations
d = np.array([1.87288, 1.23950, 0.53672, 1.29273, 1.49353])

# Plot data
ax = plt.gca()
plt.scatter(A[:,0], A[:,1])
```

```
ax.set_xlim([0, 4])
ax.set_ylim([0, 4])
ax.set_aspect('equal')
plt.show()

print(A)
```




```
[[1.8 2.5]
 [2.  1.7]
 [1.5 1.5]
 [1.5 2. ]
 [2.5 1.5]]
```

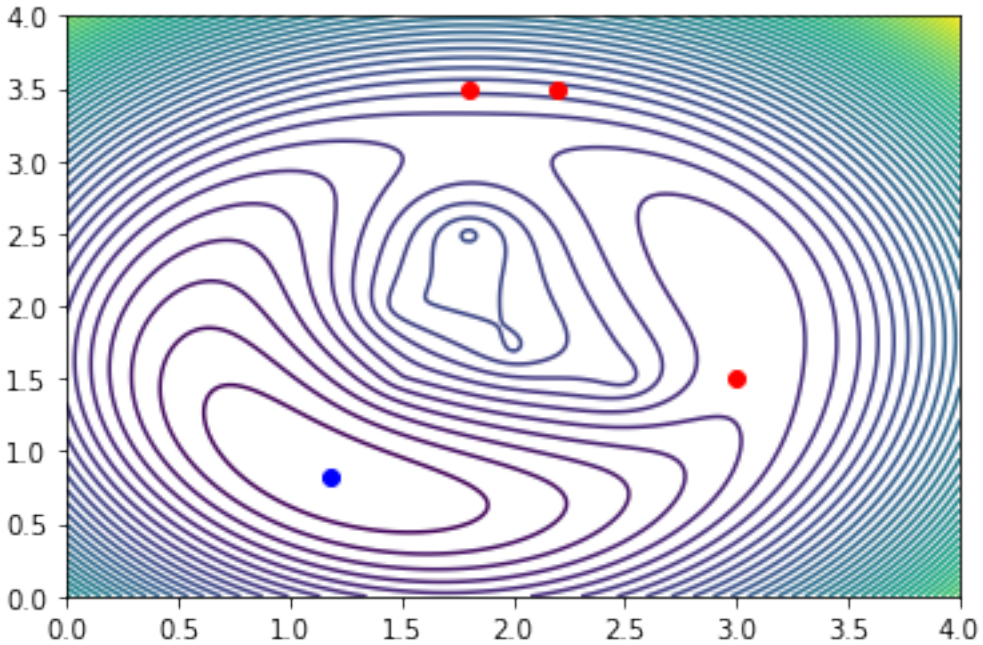
- plot the contour lines of the objective

[386]:
```
def f(x):
    f_=np.zeros(5)
    for i in range (0,len(d)):
        f_[i] = (la.norm(x-A[i])-d[i])
    return (la.norm(f_)**2)
def r(x):
    r_=np.zeros(5)
    for i in range (0,len(d)):
        r_[i] = (la.norm(x-A[i])-d[i])
    return (r_)
def Df(x):
```

```
    jac= np.zeros((5,2))
    for i in range (0,len(d)):
        jac[i] = (x-A[i])/la.norm(x-A[i])
    return jac
```

[387]:
```
import matplotlib.pyplot as plt
x=np.linspace(0,4,300)
y=np.linspace(0,4,300)
x,y=np.meshgrid(x,y)

fun = np.zeros((x.shape))
for i in range(0,len(x)):
    for j in range(0,len(x[0])):
        x0=[x[i,j],y[i,j]]
        fun[i,j]= f(x0)

plt.contour(x,y,fun,50)
plt.plot(1.8, 3.5,'ro')
plt.plot(3.0, 1.5,'ro')
plt.plot(2.2, 3.5,'ro')
plt.plot(1.18248598, 0.82422894,'bo')
```

[387]:
```
[<matplotlib.lines.Line2D at 0x7ffaa3ebf190>]
```



- Solve the problem starting from the 3 different initial points.

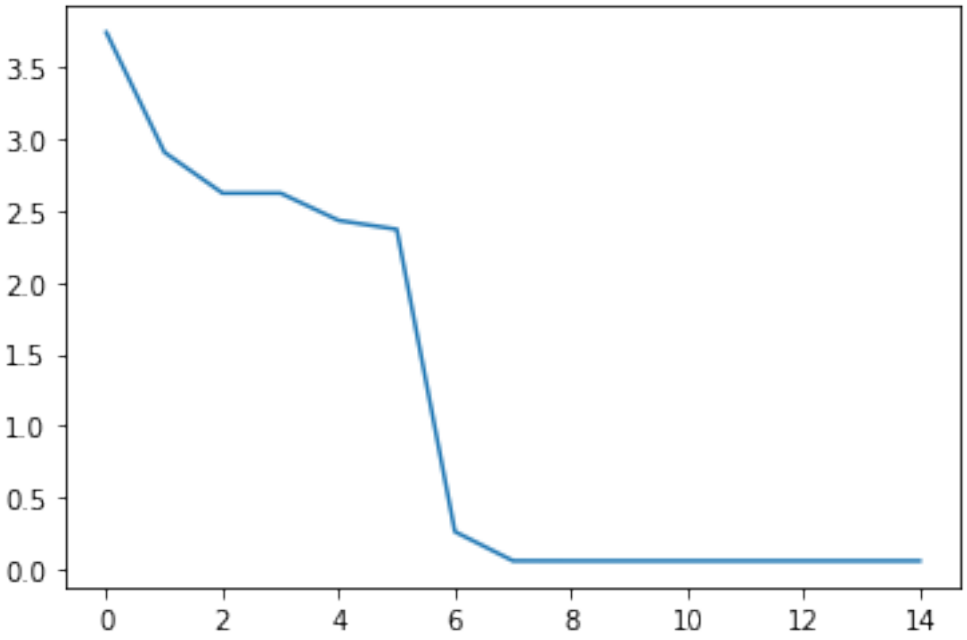
[408]:
```
x1=[1.8, 3.5]
lambda0= 0.1
x,obj,res,lam=levenberg_marquardt(f,r, Df, x1, lambda0, kmax=100, tol=1e-6)
print("x:",x)
print("obj1:",obj)
print("res1:",res)
print("lambda:",lam)
```

```
x: [1.18248598 0.82422894]
obj1: [[3.74311286]
 [2.90756626]
 [2.62261581]
 [2.62261581]
 [2.43235968]
 [2.3704844 ]
 [0.26278002]
 [0.05953178]
 [0.05911876]
 [0.05911474]
 [0.05911461]
 [0.0591146 ]
 [0.0591146 ]
 [0.0591146 ]
 [0.0591146 ]]
res1: [[3.97422889e+00]
 [3.41060526e-01]
 [1.37260887e+00]
 [1.37260887e+00]
 [4.74028290e+00]
 [6.44110737e+00]
 [1.92566902e+00]
 [7.69380091e-02]
 [3.36946000e-03]
 [5.99308996e-04]
 [1.24804433e-04]
 [2.88688167e-05]
 [7.07239371e-06]
 [1.81665848e-06]
 [4.83935927e-07]]
lambda: [[0.1       ]
 [0.08      ]
 [0.064     ]
 [0.128     ]
 [0.1024    ]
 [0.08192   ]
 [0.065536  ]
 [0.0524288 ]
```

```
[0.04194304]
[0.03355443]
[0.02684355]
[0.02147484]
[0.01717987]
[0.0137439 ]
[0.01099512]]
```

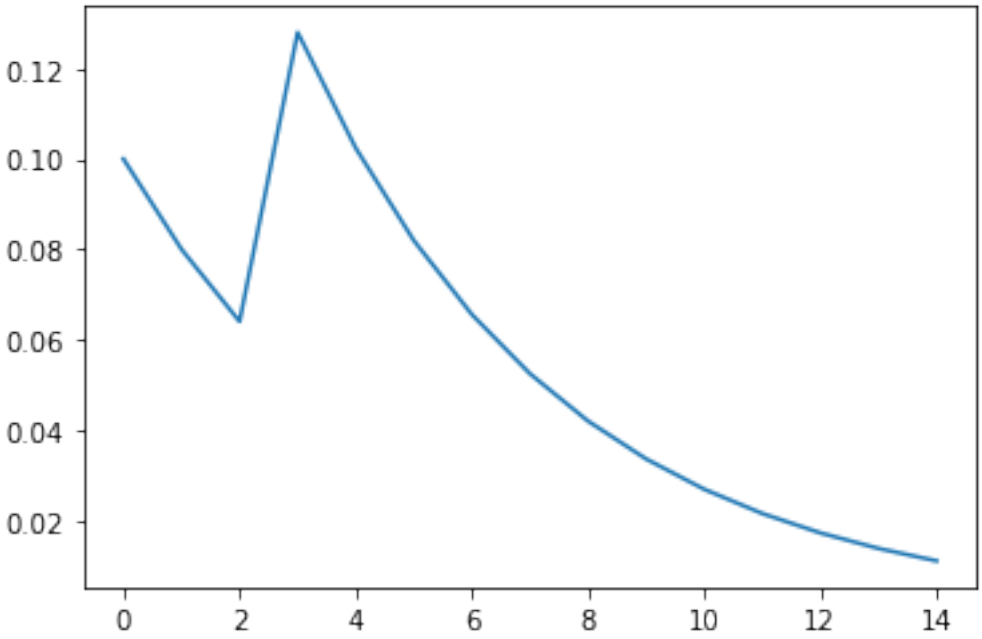
[409]:
```
plt.plot(obj)
```

[409]: [<matplotlib.lines.Line2D at 0x7ffa85fddca0>]



[410]:
```
plt.plot(lam)
```

[410]: [<matplotlib.lines.Line2D at 0x7ffa86155e50>]



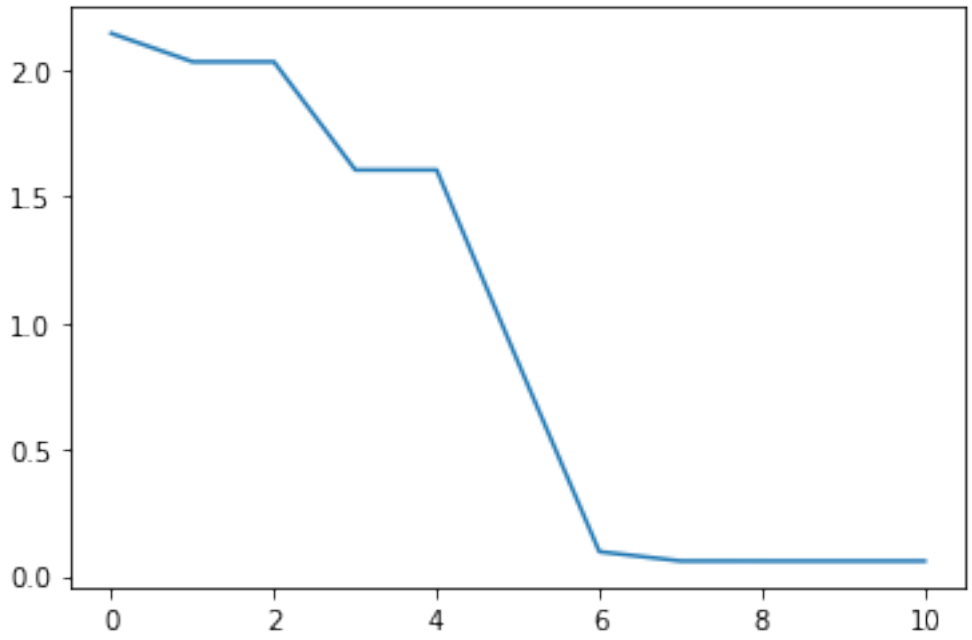
[411]:
```
x2=[3.0, 1.5]
lambda0= 0.1
x,obj,res,lam=levenberg_marquardt(f,r, Df, x2, lambda0, kmax=100, tol=1e-6)
print("x:",x)
print("obj2:",obj)
print("res2:",res)
print("lambda:",lam)
```

```
x: [1.18248579 0.82422895]
obj2: [[2.14307158]
 [2.03070847]
 [2.03070847]
 [1.60366063]
 [1.60366063]
 [0.84482321]
 [0.09596584]
 [0.05923856]
 [0.05911474]
 [0.0591146 ]
 [0.0591146 ]]
res2: [[5.18548288e-01]
 [6.71174662e-01]
 [6.71174662e-01]
 [3.25319815e+00]
 [3.25319815e+00]
 [3.51456964e+00]
```

```
 [7.97306752e-01]
 [4.52089838e-02]
 [1.57350949e-03]
 [4.19343835e-05]
 [8.22813206e-07]]
lambda: [[0.1       ]
 [0.08      ]
 [0.16      ]
 [0.128     ]
 [0.256     ]
 [0.2048    ]
 [0.16384   ]
 [0.131072  ]
 [0.1048576 ]
 [0.08388608]
 [0.06710886]]
```

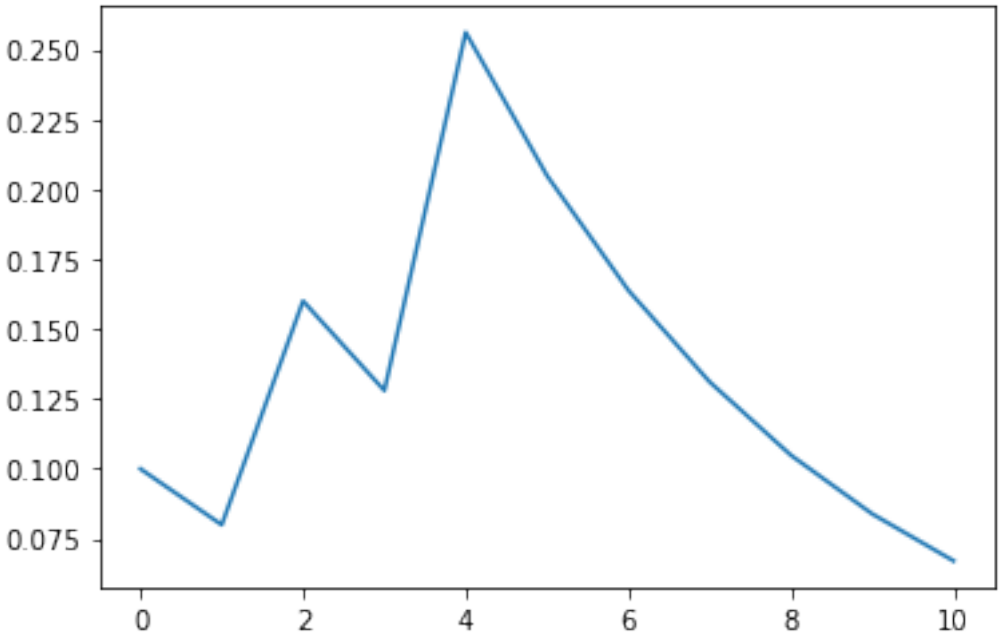
[412]:
```
plt.plot(obj)
```

[412]: [<matplotlib.lines.Line2D at 0x7ffa860f96a0>]



[413]:
```
plt.plot(lam)
```

[413]: [<matplotlib.lines.Line2D at 0x7ffa862d6340>]



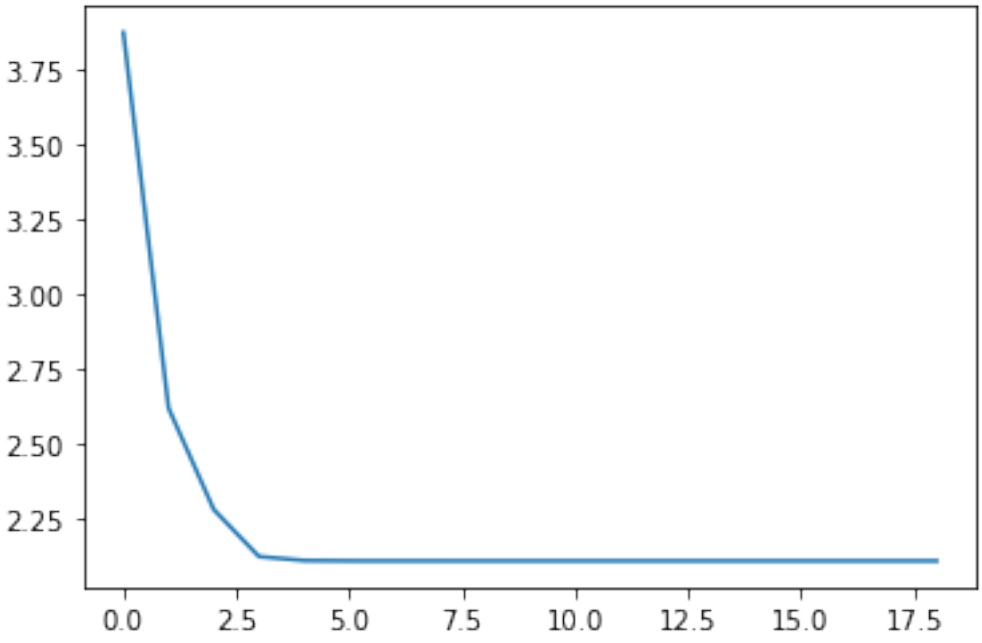
[414]:
```
x3=[2.2, 3.5]
lambda0= 0.1
x,obj,res,lam=levenberg_marquardt(f,r, Df, x3, lambda0, kmax=100, tol=1e-6)
print("x:",x)
print("obj3:",obj)
print("res3:",res)
print("lambda:",lam)
```

```
x: [2.98526641 2.1215768 ]
obj3: [[3.87469249]
 [2.62121846]
 [2.28285171]
 [2.12599578]
 [2.11264536]
 [2.11167652]
 [2.11152284]
 [2.11149093]
 [2.11148403]
 [2.11148253]
 [2.11148221]
 [2.11148214]
 [2.11148213]
 [2.11148212]
 [2.11148212]
 [2.11148212]
 [2.11148212]
```

```
 [2.11148212]
 [2.11148212]]
res3: [[4.40911490e+00]
 [1.22822422e+00]
 [1.26222345e+00]
 [3.38653631e-01]
 [5.74125436e-02]
 [1.81030794e-02]
 [8.68863486e-03]
 [4.15541900e-03]
 [1.95523101e-03]
 [9.12163448e-04]
 [4.23151224e-04]
 [1.95443651e-04]
 [8.99528606e-05]
 [4.12823154e-05]
 [1.89019047e-05]
 [8.63841097e-06]
 [3.94191556e-06]
 [1.79661224e-06]
 [8.18047837e-07]]
lambda: [[0.1       ]
 [0.08      ]
 [0.064     ]
 [0.0512    ]
 [0.04096   ]
 [0.032768  ]
 [0.0262144 ]
 [0.02097152]
 [0.01677722]
 [0.01342177]
 [0.01073742]
 [0.00858993]
 [0.00687195]
 [0.00549756]
 [0.00439805]
 [0.00351844]
 [0.00281475]
 [0.0022518 ]
 [0.00180144]]
```

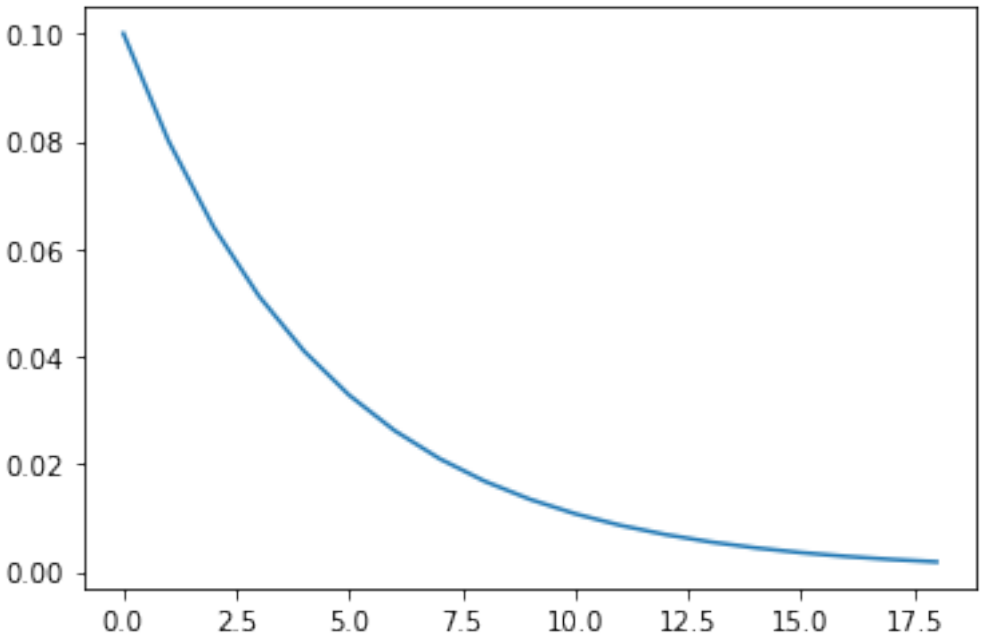
[415]:
```
plt.plot(obj)
```

[415]:
```
[<matplotlib.lines.Line2D at 0x7ffa864478b0>]
```



[416]:
```
plt.plot(lam)
```

[416]: [<matplotlib.lines.Line2D at 0x7ffa863e67f0>]



We get very differnet local minimas for differnt initial values. and notice that the behavior of

lambda. When Lambda is large, we move to the gradient dierction, otherwise, we go to Gauss-Newton direction.