Caleb Logemann
MATH 565 Continuous Optimization
Midterm Exam

1. (a) Derive an expression for the gradient $\underline{\nabla f}$.

In order to derive an expression for $\underline{\nabla f}$ we must compute all the partial derivatives $\frac{\partial f}{\partial x_k}$ and $\frac{\partial f}{\partial y_k}$ for all $1 \le k \le N$.
In order to compute these partial derivatives, I will first rewrite f in a way that seperates the x_k and y_k terms.

$$f(\underline{x}) = \left((x_k - x_k)^2 + (y_k - y_k)^2 - d_{kk}^2\right)^2 + \sum_{\substack{j=1 \\ j \neq k}}^{N} \left((x_k - x_j)^2 + (y_k - y_j)^2 - d_{kj}^2\right)^2$$
$$+ \sum_{\substack{i=1 \\ i \neq k}}^{N} \left((x_i - x_k)^2 + (y_i - y_k)^2 - d_{ik}^2\right)^2 + \sum_{\substack{j=1 \\ j \neq k}}^{N} \sum_{\substack{i=1 \\ i \neq k}}^{N} \left((x_i - x_j)^2 + (y_i - y_j)^2 - d_{ik}^2\right)^2$$

By noting that $(x_i - x_k)^2 = (x_k - x_i)^2$ and that $d_{ik} = d_{ki}$, this can be simplified to

$$f(\underline{x}) = d_{kk}^4 + 2\sum_{\substack{j=1 \\ j \neq k}}^{N} \left((x_k - x_j)^2 + (y_k - y_j)^2 - d_{kj}^2\right)^2 + \sum_{\substack{j=1 \\ j \neq k}}^{N} \sum_{\substack{i=1 \\ i \neq k}}^{N} \left((x_i - x_j)^2 + (y_i - y_j)^2 - d_{ik}^2\right)^2.$$

Now the partial derivatives of this can be taken easily note that the first and last terms don't contain x_k or y_k. The partial derivatives are

$$\frac{\partial f}{\partial x_k} = 8\sum_{\substack{j=1 \\ j \neq k}}^{N} \left(\left((x_k - x_j)^2 + (y_k - y_j)^2 - d_{kj}^2\right)(x_k - x_j)\right)$$
$$\frac{\partial f}{\partial y_k} = 8\sum_{\substack{j=1 \\ j \neq k}}^{N} \left(\left((x_k - x_j)^2 + (y_k - y_j)^2 - d_{kj}^2\right)(y_k - y_j)\right)$$

When $j = k$ the terms of these sums are zero, so these can also be expressed as

$$\frac{\partial f}{\partial x_k} = 8\sum_{j=1}^{N} \left(\left((x_k - x_j)^2 + (y_k - y_j)^2 - d_{kj}^2\right)(x_k - x_j)\right)$$
$$\frac{\partial f}{\partial y_k} = 8\sum_{j=1}^{N} \left(\left((x_k - x_j)^2 + (y_k - y_j)^2 - d_{kj}^2\right)(y_k - y_j)\right).$$

Now the gradient of f can be written using these partial derivatives, as follows

$$\underline{\nabla f(\underline{x})} = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial y_1} \\ \frac{\partial f}{\partial x_2} \\ \frac{\partial f}{\partial y_2} \\ \vdots \\ \frac{\partial f}{\partial x_N} \\ \frac{\partial f}{\partial y_N} \end{bmatrix}$$

(b) Derive an expression for the Hessian $\underline{\underline{\nabla^2 f}}$.

The Hessian for this function requires finding all possible partial second derivatives. There are several possible different partial second derivatives. They are $\frac{\partial^2 f}{\partial {x_k}^2}$, $\frac{\partial^2 f}{\partial {y_k}^2}$, $\frac{\partial^2 f}{\partial x_k \partial y_k}$, $\frac{\partial^2 f}{\partial x_k \partial x_i}$, $\frac{\partial^2 f}{\partial y_k \partial y_i}$, and $\frac{\partial^2 f}{\partial x_k \partial y_i}$, where $k \neq i$.

First I will compute $\frac{\partial^2 f}{\partial {x_k}^2}$ and $\frac{\partial^2 f}{\partial {y_k}^2}$.

$$\begin{aligned}
\frac{\partial^2 f}{\partial {x_k}^2} &= \frac{\partial}{\partial x_k}\left(8\sum_{j=1}^{N}\Big((x_k - x_j)^2 + (y_k - y_j)^2 - d_{kj}^2\Big)(x_k - x_j)\right) \\
&= \frac{\partial}{\partial x_k}\left(8\sum_{j=1}^{N}(x_k - x_j)^3 + \Big((y_k - y_j)^2 - d_{kj}^2\Big)(x_k - x_j)\right) \\
&= 8\sum_{j=1}^{N}\frac{\partial}{\partial x_k}\Big((x_k - x_j)^3\Big) + \Big((y_k - y_j)^2 - d_{kj}^2\Big)\frac{\partial}{\partial x_k}((x_k - x_j)) \\
&= 8\sum_{j=1}^{N}3(x_k - x_j)^2 + (y_k - y_j)^2 - d_{kj}^2 \\
\frac{\partial^2 f}{\partial {y_k}^2} &= \frac{\partial}{\partial y_k}\left(8\sum_{j=1}^{N}\Big((x_k - x_j)^2 + (y_k - y_j)^2 - d_{kj}^2\Big)(y_k - y_j)\right) \\
&= 8\sum_{j=1}^{N}(x_k - x_j)^2 + 3(y_k - y_j)^2 - d_{kj}^2
\end{aligned}$$

Next I will compute $\dfrac{\partial^2 f}{\partial x_k \partial y_k}$.

$$\begin{aligned}
\frac{\partial^2 f}{\partial x_k \partial y_k} &= \frac{\partial}{\partial y_k}\left(8\sum_{j=1}^{N}\Big((x_k-x_j)^2+(y_k-y_j)^2-d_{kj}^2\Big)(x_k-x_j)\right) \\
&= 8\sum_{j=1}^{N}(x_k-x_j)\frac{\partial}{\partial y_k}\Big((y_k-y_j)^2\Big) \\
&= 16\sum_{j=1}^{N}(x_k-x_j)(y_k-y_j)
\end{aligned}$$

Thirdly I will compute $\dfrac{\partial^2 f}{\partial x_k \partial x_i}$ and $\dfrac{\partial^2 f}{\partial y_k \partial y_i}$, for $i \neq k$.

$$\begin{aligned}
\frac{\partial^2 f}{\partial x_k \partial x_i} &= \frac{\partial}{\partial x_i}\left(8\sum_{j=1}^{N}\Big((x_k-x_j)^2+(y_k-y_j)^2-d_{kj}^2\Big)(x_k-x_j)\right) \\
&= \frac{\partial}{\partial x_i}\Big(8\Big((x_k-x_i)^2+(y_k-y_i)^2-d_{ki}^2\Big)(x_k-x_i)\Big) \\
&= \frac{\partial}{\partial x_i}\Big(8(x_k-x_i)^3+8\Big((y_k-y_i)^2-d_{ki}^2\Big)(x_k-x_i)\Big) \\
&= -24(x_k-x_i)^2-8\Big((y_k-y_i)^2-d_{ki}^2\Big) \\
&= -8\Big((x_k-x_i)^2+(y_k-y_i)^2-d_{ki}^2\Big) \\
\frac{\partial^2 f}{\partial y_k \partial y_i} &= \frac{\partial}{\partial y_i}\left(8\sum_{j=1}^{N}\Big((x_k-x_j)^2+(y_k-y_j)^2-d_{kj}^2\Big)(y_k-y_j)\right) \\
&= \frac{\partial}{\partial y_i}\Big(8\Big((x_k-x_i)^2+(y_k-y_i)^2-d_{ki}^2\Big)(y_k-y_i)\Big) \\
&= \frac{\partial}{\partial y_i}\Big(8\Big((x_k-x_i)^2-d_{ki}^2\Big)(y_k-y_i)+(y_k-y_i)^3\Big) \\
&= -8\Big((x_k-x_i)^2+3(y_k-y_i)^2-d_{ki}^2\Big)
\end{aligned}$$

Lastly I will compute $\dfrac{\partial^2 f}{\partial x_k \partial y_i}$ for $i \neq k$. Note that this is equivalent to $\dfrac{\partial^2 f}{\partial y_i \partial x_k}$.

$$\begin{aligned}
\frac{\partial^2 f}{\partial x_k \partial y_i} &= \frac{\partial}{\partial y_i}\left(8\sum_{j=1}^{N}\Big((x_k-x_j)^2+(y_k-y_j)^2-d_{kj}^2\Big)(x_k-x_j)\right) \\
&= \frac{\partial}{\partial y_i}\Big(8\Big((x_k-x_i)^2+(y_k-y_i)^2-d_{ki}^2\Big)(x_k-x_i)\Big) \\
&= -16(x_k-x_i)(y_k-y_i)
\end{aligned}$$

Now with all these partial derivatives the Hessian can be shown as a matrix of partial derivatives.

$$\underline{\underline{\nabla^2 f}} = \begin{bmatrix} \frac{\partial^2 f}{\partial {x_1}^2} & \frac{\partial^2 f}{\partial x_1 \partial y_1} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_N} & \frac{\partial^2 f}{\partial x_1 \partial y_N} \\ \frac{\partial^2 f}{\partial y_1 \partial x_1} & \frac{\partial^2 f}{\partial {y_1}^2} & \cdots & \frac{\partial^2 f}{\partial y_1 \partial x_N} & \frac{\partial^2 f}{\partial y_1 \partial y_N} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \frac{\partial^2 f}{\partial x_N \partial x_1} & \frac{\partial^2 f}{\partial x_N \partial y_1} & \cdots & \frac{\partial^2 f}{\partial {x_N}^2} & \frac{\partial^2 f}{\partial x_N \partial y_N} \\ \frac{\partial^2 f}{\partial y_N \partial x_1} & \frac{\partial^2 f}{\partial y_N \partial y_1} & \cdots & \frac{\partial^2 f}{\partial y_N \partial x_N} & \frac{\partial^2 f}{\partial {y_N}^2} \end{bmatrix}$$

(c) Prove that the Hessian, $\underline{\underline{\nabla^2 f}}$ is symmetric positive definite for all $\underline{x} \in \mathbb{R}^{2N}$.

Proof. Cleary the Hessian is symmetric as all of the partial derivatives are symmetric. I don't believe that the Hessian is in face positive definite or positive semidefinite. In order to show this consider the situation with $N = 2$ that is only two cities. In this case the f can be written as

$$f(\underline{x}) = 2((x_1 - x_2)^2 + (y_1 - y_2)^2 - d_{12}^2)^2$$

I will evaluate the Hessian at $\underline{x} = [0, 0, 0, 0]^T$. In this case the partial derivatives become

$$\begin{aligned} \frac{\partial^2 f}{\partial {x_k}^2} &= 8 \sum_{j=1}^{2} 3(x_k - x_j)^2 + (y_k - y_j)^2 - d_{kj}^2 \\ &= -8d_{12}^2 \\ \frac{\partial^2 f}{\partial {y_k}^2} &= 8 \sum_{j=1}^{2} (x_k - x_j)^2 + 3(y_k - y_j)^2 - d_{kj}^2 \\ &= -8d_{12}^2 \\ \frac{\partial^2 f}{\partial x_k \partial y_k} &= 16 \sum_{j=1}^{2} (x_k - x_j)(y_k - y_j) \\ &= 0 \\ \frac{\partial^2 f}{\partial x_k \partial x_i} &= -8\Big((x_k - x_i)^2 + (y_k - y_i)^2 - d_{ki}^2\Big) \\ &= 8d_{12}^2 \\ \frac{\partial^2 f}{\partial y_k \partial y_i} &= -8\Big((x_k - x_i)^2 + 3(y_k - y_i)^2 - d_{ki}^2\Big) \\ &= 8d_{12}^2 \end{aligned}$$

Using part these the Hessian is

$$\underline{\underline{\nabla^2 f}} = \begin{bmatrix} -8d_{12}^2 & 0 & 8d_{12}^2 & 0 \\ 0 & -8d_{12}^2 & 0 & 8d_{12}^2 \\ 8d_{12}^2 & 0 & -8d_{12}^2 & 0 \\ 0 & 8d_{12}^2 & 0 & -8d_{12}^2 \end{bmatrix}$$

The eigenvalues of this matrix are $-16d_{12}^2$ and 0 both of multiplicity 2. To see this let $\underline{u_1} =$

$[1, 1, 1, 1]^T$, then $\underline{\underline{\nabla^2 f}}\,\underline{u_1}$ is

$$\begin{bmatrix} -8d_{12}^2 & 0 & 8d_{12}^2 & 0 \\ 0 & -8d_{12}^2 & 0 & 8d_{12}^2 \\ 8d_{12}^2 & 0 & -8d_{12}^2 & 0 \\ 0 & 8d_{12}^2 & 0 & -8d_{12}^2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

This is also true for $\underline{u_2} = [1, -1, 1, -1]^T$, so $\underline{u_1}$ and $\underline{u_2}$ are linearly independent eigenvectors with eigenvalue 0.

Consider $\underline{u_3} = [-1, -1, 1, 1]$ and $\underline{u_4} = [1, -1, -1, 1]$.

$$\begin{bmatrix} -8d_{12}^2 & 0 & 8d_{12}^2 & 0 \\ 0 & -8d_{12}^2 & 0 & 8d_{12}^2 \\ 8d_{12}^2 & 0 & -8d_{12}^2 & 0 \\ 0 & 8d_{12}^2 & 0 & -8d_{12}^2 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 16d_{12}^2 \\ 16d_{12}^2 \\ -16d_{12}^2 \\ -16d_{12}^2 \end{bmatrix} = -16d_{12}^2 \underline{u_3}$$

This shows that $\underline{u_3}$ and $\underline{u_4}$ are linearly independent eigenvectors of $\underline{\underline{\nabla^2 f}}$ with eigenvalue $-16d_{12}^2$. These are negative eigenvalues, so this is an example of a vector in $\mathbb{R}^{2N}$ where $\underline{\underline{\nabla^2 f}}$ is not positive definite or even positive semidefinite. $\square$

2. (a) The following function implement the Trust Region Method.

```
import numpy as np
import ipdb
def trustRegionMethod(f, gradf, B, minimizingFunction, x0, maxDelta, Delta0, eta,
    ↪ TOL, MaxIter):
    N = len(x0)
    x = np.zeros([MaxIter+1, N])
    x[0] = x0
    nIter = 0
    mstop = 1
    Delta = Delta0

    gradfx = gradf(x[nIter])
    while nIter < MaxIter and mstop:
        Bx = B(x[nIter])
        p = minimizingFunction(gradfx, Bx, Delta)

        m = lambda p: f(x[nIter]) + np.dot(p, gradfx) + (1.0/2.0)*np.dot(p, np.dot(
            ↪ Bx, p))
        deltaM = m(np.zeros(N)) - m(p)
        rho = (f(x[nIter]) - f(x[nIter] + p))/(deltaM)

        # modify trust region size
        if rho < 1.0/4.0:
            Delta = (1.0/4.0)*Delta
        elif rho > 3.0/4.0 and abs(np.linalg.norm(p) - Delta) < 1e-5:
            Delta = min(2*Delta, maxDelta)

        # Decide whether or not to reject step
        if rho > eta:
            x[nIter + 1] = x[nIter] + p
        else:
            x[nIter + 1] = x[nIter]

        nIter+=1
```

```
        if nIter % 100 == 0:
            print((nIter + 0.0)/MaxIter)
        gradfx = gradf(x[nIter])
        if np.linalg.norm(p) < TOL and np.linalg.norm(gradfx) < TOL:
            mstop = 0

    return x[:nIter+1]
```

(b) The following function implements the Dogleg Method. This is a variant which does the Cauchy point method instead of the Dogleg Method when the matrix B is not symmetric positive definite.

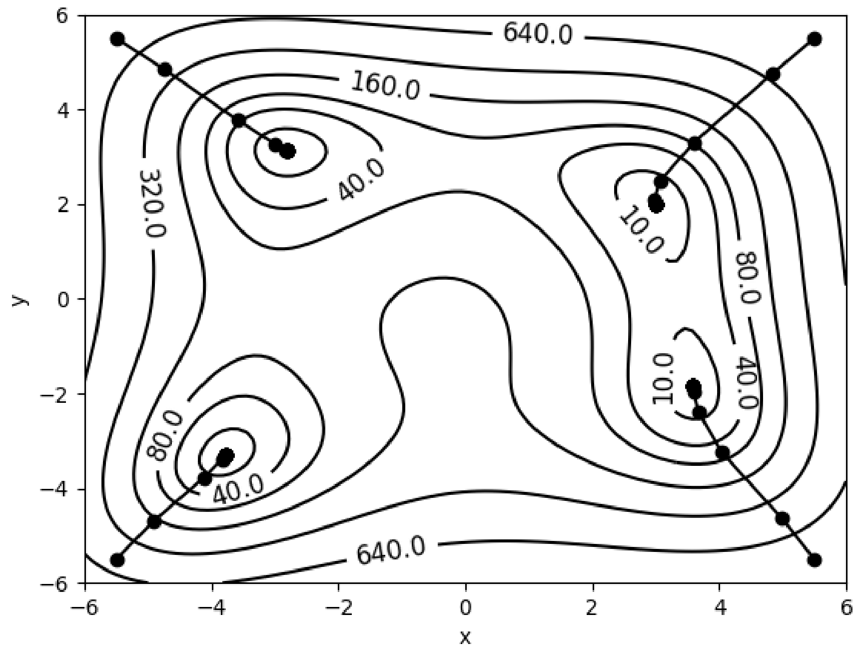
```
import numpy as np
def doglegMethod(g, B, Delta):
    # exact minimizer of approximation function, m
    pB = -np.linalg.solve(B, g)
    if np.linalg.norm(pB) < Delta:
        return pB

    # steepest descent minimizer
    pU = -np.dot(g, g)/np.dot(g, np.dot(B, g)) * g
    if np.linalg.norm(pU) > Delta:
        return Delta/np.linalg.norm(pU) * pU

    # use dogleg path
    # solve (pU + alpha(pB - pU))^T (pU + alpha(pB - pU)) = Delta^2
    a = np.dot(pB - pU, pB - pU)
    b = 2*np.dot(pU, pB - pU)
    c = np.dot(pU, pU) - Delta**2

    alpha0 = (-b + sqrt(b**2 - 4*a*c))/(2*a)
    alpha1 = (-b - sqrt(b**2 - 4*a*c))/(2*a)

    if alpha0 > 0 and alpha0 < 1:
        return pU + alpha0*(pB - pU)
    else:
        return pU + alpha1*(pB - pU)
```

(c) The following scipt uses the previous two functions to minimize f.

```
import numpy as np
execfile("trustRegionMethod.py")
execfile("doglegMethod.py")
f = lambda x: (x[0]**2 + x[1] - 11)**2 + (x[0] + x[1]**2 - 7)**2

def gradf(x):
    dx0 = 4*(x[0]**3 + x[0]*(x[1] - 11)) + 2*(x[0] + x[1]**2 - 7)
    dx1 = 2*(x[0]**2 + x[1] - 11) + 4*(x[1]**3 + x[1]*(x[0] - 7))
    return np.array([dx0, dx1])

def hessianf(x):
    dx0x0 = 12*x[0]**2 + 4*x[1] - 42
    dx0x1 = 4*x[0] + 4*x[1]
    dx1x1 = 12*x[1]**2 + 3*x[0] - 26
    return np.array([[dx0x0, dx0x1],[dx0x1, dx1x1]])

def plotResults(f, x0Range, x1Range, solList, title):
    minX0 = x0Range[0]
    maxX0 = x0Range[1]
```

```
    minX1 = x1Range[0]
    maxX1 = x1Range[1]
    meshSize = 100

    x0list = np.linspace(minX0, maxX0, meshSize)
    x1list = np.linspace(minX1, maxX1, meshSize)
    X0, X1 = np.meshgrid(x0list, x1list)
    Z = np.zeros([meshSize,meshSize])
    for i in range(meshSize):
        for j in range(meshSize):
            Z[i,j] = f([X0[i,j], X1[i,j]])

    plt.figure()
    levels = [10, 40, 80, 160, 320, 640]
    contour = plt.contour(X0, X1, Z, levels, colors='k')
    for i in range(len(solList)):
        plt.plot(solList[i][:,0], solList[i][:,1], '-k')
        plt.plot(solList[i][:,0], solList[i][:,1], 'ko')
    plt.clabel(contour, colors='k', fmt='%2.1f', fontsize=12)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title(title)
    plt.show()

maxDelta = 2
Delta0 = 1

eta = .2
TOL = 1e-10
MaxIter = 100

x0 = [5.5, 5.5]
sol0 = trustRegionMethod(f, gradf, hessianf, doglegMethod, x0, maxDelta, Delta0,
    ↪ eta, TOL, MaxIter)
x0 = [-5.5, 5.5]
sol1 = trustRegionMethod(f, gradf, hessianf, doglegMethod, x0, maxDelta, Delta0,
    ↪ eta, TOL, MaxIter)
x0 = [5.5, -5.5]
sol2 = trustRegionMethod(f, gradf, hessianf, doglegMethod, x0, maxDelta, Delta0,
    ↪ eta, TOL, MaxIter)
x0 = [-5.5, -5.5]
sol3 = trustRegionMethod(f, gradf, hessianf, doglegMethod, x0, maxDelta, Delta0,
    ↪ eta, TOL, MaxIter)
solList = [sol0, sol1, sol2, sol3]

plotResults(f, [-6, 6], [-6, 6], solList, "")
```

I used a tolerance of 10^{-10}. The initial guesses that I used where $(5.5, 5.5)$, $(-5.5, 5.5)$, $(5.5, -5.5)$, and $(-5.5, 5.5)$. The number of iterations that for each initial guess were 16, 12, 17, and 13 respectively. The final positions were $(3, 2)$, $(-2.8, 3.13)$, $(3.58, -1.85)$, and $(-3.78, -3.28)$. The value of f at all of these points was 0. The script also gave the following image.



3. I used the following script to try and solve this problem. This script used the Cauchy Point method instead of the Dogleg because the Hessian is not always positive definite.

```
execfile("trustRegionMethod.py")
execfile("doglegMethod.py")
execfile("cauchyPointMethod.py")

d = np.array([[   0,  587, 1212,  701, 1936,  604,  748, 2139, 2182,  543,  762],
              [ 587,    0,  920,  940, 1745, 1188,  713, 1858, 1737,  597,  309],
              [1212,  920,    0,  879,  831, 1726, 1631,  949, 1021, 1494,  614],
              [ 701,  940,  879,    0, 1374,  968, 1420, 1645, 1891, 1220,  854],
              [1936, 1745,  831, 1374,    0, 2339, 2451,  347,  959, 2300, 1443],
              [ 604, 1188, 1726,  968, 2339,    0, 1092, 2594, 2734,  923, 1361],
              [ 748,  713, 1631, 1420, 2451, 1092,    0, 2571, 2408,  205, 1021],
              [2139, 1858,  949, 1645,  347, 2594, 2571,    0,  678, 2442, 1548],
              [2182, 1737, 1021, 1891,  959, 2734, 2408,  678,    0, 2329, 1451],
              [ 543,  597, 1494, 1220, 2300,  923,  205, 2442, 2329,    0,  898],
              [ 762,  309,  614,  854, 1443, 1361, 1021, 1548, 1451,  898,    0]])
N = len(d)
def f(v):
    x = v[::2]
    y = v[1::2]
    return sum([sum([((x[i] - x[j])**2 + (y[i] - y[j])**2 - d[i,j]**2)**2 for j in
        ↪ range(N)]) for i in range(N)])

def gradf(x):
    gradfx = np.zeros(2*N)
    for k in range(N):
        # \pd{f}{x_k}
        gradfx[2*k] = 8*sum([((x[2*k] - x[2*j])**2 + (x[2*k+1] - x[2*j+1])**2 - d[k,j
            ↪ ]**2)*(x[2*k] - x[2*j]) for j in range(N)])
        # \pd{f}{y_k}
        gradfx[2*k+1] = 8*sum([((x[2*k] - x[2*j])**2 + (x[2*k+1] - x[2*j+1])**2 - d[k,j
            ↪ ]**2)*(x[2*k+1] - x[2*j+1]) for j in range(N)])
```

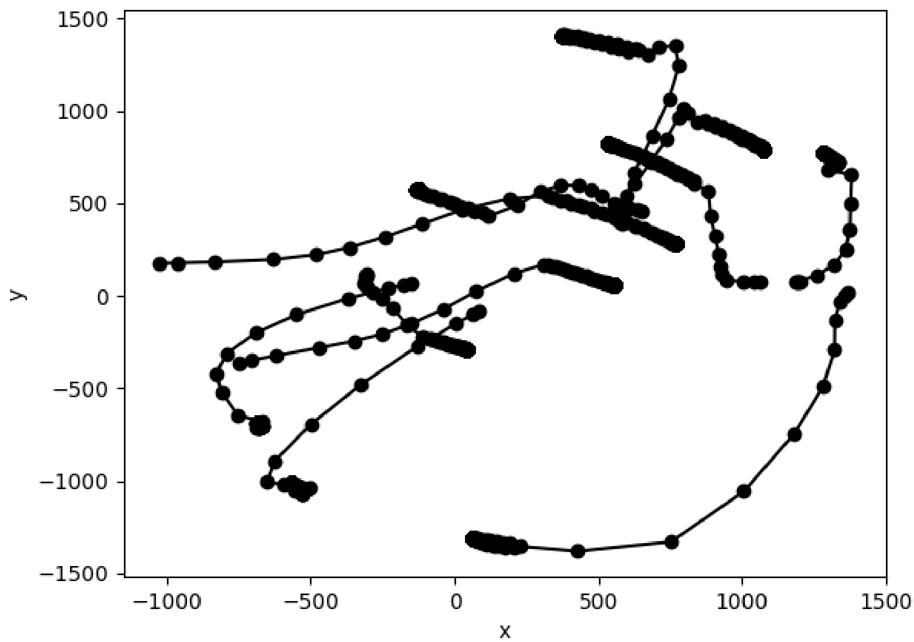
```
    return gradfx

def hessianf(x):
    hessianfx = np.zeros([2*N, 2*N])
    for k in range(N):
        for i in range(N):
            if i == k:
                # \pd[2]{f}{x_k x_k}
                hessianfx[2*k, 2*k] = 8*sum([3*(x[2*k] - x[2*j])**2 + (x[2*k+1] - x[2*j+1])**2 - d[k,j]**2 for j in range(N)])
                # \pd[2]{f}{x_k y_k}
                hessianfx[2*k, 2*k+1] = 16*sum([(x[2*k] - x[2*j])*(x[2*k+1] - x[2*j+1]) for j in range(N)])
            else:
                # \pd[2]{f}{x_k x_i}
                hessianfx[2*k, 2*i] = -8*(3*(x[2*k] - x[2*i])**2 + (x[2*k+1] - x[2*i+1])**2 - d[k,i]**2)
                # \pd[2]{f}{x_k y_i}
                hessianfx[2*k, 2*i+1] = -16*(x[2*k+1] - x[2*i+1])*(x[2*k] - x[2*i])

        for i in range(N):
            if i == k:
                # \pd[2]{f}{y_k x_k}
                hessianfx[2*k+1, 2*k] = 16*sum([(x[2*k] - x[2*j])*(x[2*k+1] - x[2*j+1]) for j in range(N)])
                # \pd[2]{f}{y_k y_k}
                hessianfx[2*k+1, 2*k+1] = 8*sum([(x[2*k] - x[2*j])**2 + 3*(x[2*k+1] - x[2*j+1])**2 - d[k,j]**2 for j in range(N)])
            else:
                # \pd[2]{f}{y_k x_i}
                hessianfx[2*k+1, 2*i] = -16*(x[2*k+1] - x[2*i + 1])*(x[2*k] - x[2*i])
                # \pd[2]{f}{y_k y_i}
                hessianfx[2*k+1, 2*i+1] = -8*((x[2*k] - x[2*i])**2 + 3*(x[2*k+1] - x[2*i+1])**2 - d[k,i]**2)
    return hessianfx

def hessianf2(v):
    hessianfx = np.zeros([2*N, 2*N])
    x = v[::2]
    y = v[1::2]
    for m in range(2*N):
        k = m/2
        for n in range(2*N):
            i = n/2
            # derivative with respect to x_k
            if m % 2 == 0:
                # derivative with respect to x_i
                if n % 2 == 0:
                    # \pd[2]{f}{x_k}
                    if k == i:
                        hessianfx[m,n] = 8*sum([(3*(x[k] - x[j])**2 + (y[k] - y[j])**2 - d[k,j]**2) for j in range(N)])
                    # \mpd[2]{f}{\partial x_k \partial x_i}
                    else:
                        hessianfx[m,n] = -8*(3*(x[k] - x[i])**2 + (y[k] - y[i])**2 - d[k,i]**2)
                # derivative with respect to y_i
                else:
                    # \mpd[2]{f}{\partial x_k \partial y_k}
                    if k == i:
                        hessianfx[m, n] = 16*sum([(x[k] - x[j])*(y[k] - y[j]) for j in
```

```
                            ↪ range(N)])
                    # \mpd[2]{f}{\partial x_k \partial y_i}
                    else:
                        hessianfx[m, n] = -16*(y[k] - y[i])*(x[k] - x[i])
            # derivative with respect to y_k
            else:
                # derivative with respect to x_i
                if n % 2 == 0:
                    # \mpd[2]{f}{\partial y_k \partial x_k}
                    if k == i:
                        hessianfx[m, n] = 16*sum([(x[k] - x[j])*(y[k] - y[j]) for j in
                            ↪ range(N)])
                    # \mpd[2]{f}{\partial y_k \partial x_i}
                    else:
                        hessianfx[m, n] = -16*(x[k] - x[i])*(y[k] - y[i])
                # derivative with respect to y_i
                else:
                    # \pd[2]{f}{y_k}
                    if k == i:
                        hessianfx[m, n] = 8*sum([((x[k] - x[j])**2 + 3*(y[k] - y[j])**2
                            ↪  - d[k,j]**2) for j in range(N)])
                    # \mpd[2]{f}{\partial y_k \partial y_i}
                    else:
                        hessianfx[m, n] = -8*((x[k] - x[i])**2 + 3*(y[k] - y[i])**2 - d
                            ↪ [k,i]**2)
    return hessianfx

def plotResults(sol, title):
    plt.figure()
    for i in range(N):
        plt.plot(sol[:,2*i], sol[:,2*i+1], '-k')
        plt.plot(sol[:,2*i], sol[:,2*i+1], 'ko')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title(title)
    plt.show()

maxDelta = 10
Delta0 = 10

eta = 0
TOL = 1e-10
MaxIter = 5000

x0 = np.zeros(2*N)
xCoordinates = 3000*np.random.rand(N) - 1500
yCoordinates = 1000*np.random.rand(N) - 500
x0[::2] = xCoordinates
x0[1::2] = yCoordinates

sol = trustRegionMethod(f, gradf, hessianf, cauchyPointMethod, x0, maxDelta, Delta0,
    ↪ eta, TOL, MaxIter)

plotResults(sol, "")
```

```
import numpy as np
def cauchyPointMethod(g, B, Delta):
    gBg = np.dot(g, np.dot(B, g))
    if gBg <= 0:
        tau = 1
```

```
    else:
        tau = min(1, np.linalg.norm(g)**3/(Delta*gBg))
    return (-tau*Delta/np.linalg.norm(g))*g
```

The following image was output.



My initial guess is described by the following vector.

```
array([ 1063.13361662,    74.46012024, -1028.17368688,   176.50472984,
    -307.20176908,   114.15633047,   653.7460869 ,   459.86440728,
    -149.77434281,    66.42794805,   595.42644861,   472.62825164,
    1201.24896009,    73.58942836,    86.98095451,   -83.02657181,
    1369.98306723,    17.08922723,   582.09891979,   392.28165874,
    -748.20081431,  -361.17194912])
```

My final position was

```
array([  532.61219259,   821.19453455,   767.60659155,   280.21998863,
     44.6826028 ,  -289.15134043,  -125.32556221,   572.57935113,
   -685.19352323,  -684.78195294,   378.01600111,  1403.31655501,
   1280.25315662,   774.24900118,  -565.03254695, -1014.60503282,
     61.34327137, -1307.75659244,  1076.09321237,   791.56902424,
    554.21204466,    55.97004481])
```

The value of my function was 7909452167.470356. The method did not converge in 3000 steps. This seems pretty high, but I was unable to find a better solution. I believe that my final answer is pretty reasonable, and I think that maybe I can identify the different cities, but I am not very satisfied with this solution.