Numerical methods and Algorithms Engineering Project 2 report

E7650 (Task 4) Valentin SERVIERES P170B115

For this Engineering project, we will be using these libraries from python.

```
import numpy as np
import matplotlib.pyplot as plt
import math
from matplotlib import cm
from mpl_toolkits import mplot3d
```

1 Linear equation systems : Gaussian algorithm

For this task, I had to solve the system of equations below using Gaussian algorithm.

$$\begin{cases} 3x_1 + 7x_2 + x_3 + 3x_4 = 40 \\ x_1 - 6x_2 + 6x_3 + 8x_4 = 19 \\ 4x_1 + 4x_2 - 7x_3 + x_4 = 36 \\ 4x_1 + 16x_2 + 2x_3 = 48 \end{cases} \tag{1}$$

So, to be able to use Gaussian algorithm we will have to transform this equation system to an augmented matrix. An augmented matrix contains the values of the coeficients we are looking for (x_1, x_2, x_3, x_4) on the left side and the value associated to each line on the right side.

```
A4=np.matrix([[3.0 , 7.0,  1.0,  3.0],
              [1.0, -6.0,  6.0,  8.0],
              [4.0,  4.0, -7.0,  1.0],
              [4.0,  16.0,  2.0,  0.0]])

b1=(np.matrix([40,
               19,
               36,
               48])).transpose()

A = A4
b = b4

nbeq=(np.shape(A))[0]   # number of
    equations
nbsol=(np.shape(b))[1]  # number of
    solutions

expA=np.hstack((A,b))  # expanding
    matrix
```

$$\left[\begin{array}{cccc|c} 3 & 7 & 1 & 3 & 40 \\ 1 & -6 & 6 & 8 & 19 \\ 4 & 4 & -7 & 1 & 36 \\ 4 & 16 & 2 & 0 & 48 \end{array}\right]$$

Figure 1: Code and Mathematical representations of this system of equations

The Gaussian algorithm has to handle infinite solution cases, and no solution cases. These cases are handled in the second part of the Gaussian algorithm (during the backward steps).
For the first one we just take an arbitrary value (0 in our case) to x_i. This case appears when a whole

line of the augmented matrix is full of 0 (indeed, $x_i \times 0 = 0, \forall x_i \in \mathbb{R}$).
For the second case we stop the algorithm when the left side of the augmented matrix has only 0, but the right side has a different value from 0. It is the case of "No solution".
Also, this algorithm needs to be able to swap lines of the augmented matrix to always keep a pivoting element different from 0.

```
def swapLines(A, i, n):
    # print("swiping line:", i)
    maxCoef=max(abs(A[i:n,i]));
    imax=abs(A[i:n,i]).argmax()
    if abs(maxCoef) > 0:
        A[[i,i+imax],:]=A[[i+imax,i],:] # switching lines with the one which has the maximum value
        return [1, A]
    else:
        return [0, A]
```

Here is the Gaussian algorithm:

```
def gaussianSolver(expA, nbeq, nbsol):
    # forward steps
    for i in range (0,nbeq-1):   # range starts at 0 finish at nbeq-2
        if expA[i,i] == 0:
            swipeResult = swapLines(expA, i, nbeq)
            if swipeResult[0] == 0: # no line have been swapped
                print("Infinite solutions")
            else:
                expA = swipeResult[1]
        for j in range (i+1,nbeq):  # range starts i+1 and finish at nbeq-1
            if abs(expA[i,i]) > 1e-8:
                expA[j,i:nbeq+nbsol]=expA[j,i:nbeq+nbsol]-expA[i,i:nbeq+nbsol]*expA[j,i]/e
                expA[j,i]=0

    # backward steps:
    sols=np.zeros(shape=(nbeq,nbsol))
    for i in range (nbeq-1,-1,-1):    # range starts at nbeq-1 and finish 0 (third param is step)
        # we can have some values like 1e-15 that are not considered as 0 whereas it should
        if abs(expA[i, :nbeq-1].any()) < 1e-8 and abs(expA[i, nbeq]) < 1e-8:
            print ("infinite x" + str(i+1) + " solutions, taking 0")
        elif abs(expA[i, :nbeq].any()) < 1e-8 and abs(expA[i, nbeq]) > 1e-8:
            return "No solution for this system"
        else:
            sols[i,:]=(expA[i,nbeq:nbeq+nbsol]-expA[i,i+1:nbeq]*sols[i+1:nbeq,:])/expA[i,i
    return sols
```

Finally, the only thing that we have to do is call this function:

```
solutions = gaussianSolver(expA, nbeq, nbsol)
print(solutions)
```

For the tests, I used the first 5 equations of the instructions' table 1. To verify the results, I used the website `https://www.WolframAlpha.com/`.

1.1 Equation 1

```
A1=np.matrix([[1.0, -2.0,  3.0, 4.0],
              [1.0,  0.0, -1.0, 1.0],
              [2.0, -2.0,  2.0, 5.0],
              [0.0, -7.0,  3.0, 1.0]])

```

```
b1=(np.matrix([11,
               -4,
               7,
               2])).transpose()
```

We obtain an infinite number of solutions :

```
infinite x4 solutions, taking 0
[[0.59090909]
[1.68181818]
[4.59090909]
[0.]]
```

Verification: we see that if we take $a = 0$ we have the same solutions

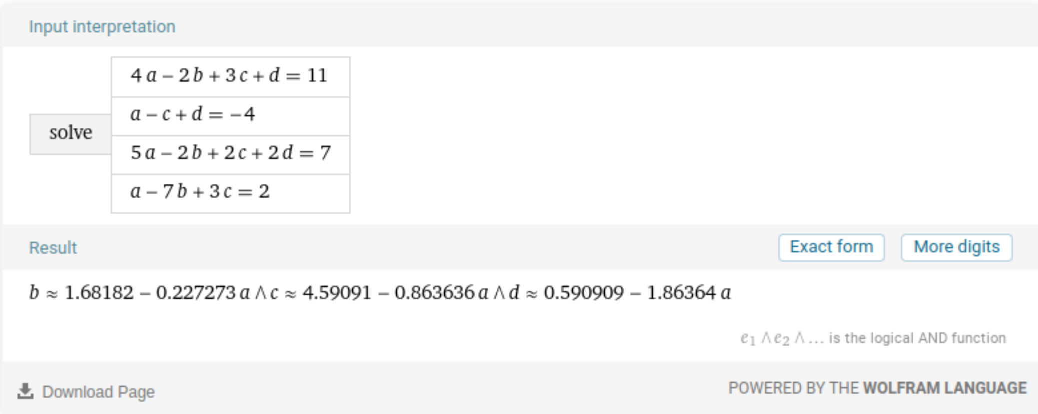

Figure 2: Results from WolframAlpha

1.2 Equation 2

```
A2=np.matrix([[3.0,  7.0,   1.0,   3.0],
              [1.0, -6.0,   6.0,   9.0],
              [4.0,  4.0,  -7.0,   1.0],
              [-1.0, 3.0,   8.0,   2.0]])

b2=(np.matrix([37,
               11,
               38,
               -1])).transpose()
```

We obtain an infinite number of solutions :

```
infinite x4 solutions, taking 0 [[10.27759197]
[0.75585284]
[0.87625418]
[0.]]
```

Verification: we see that if we take $a = 0$ we have the same solutions

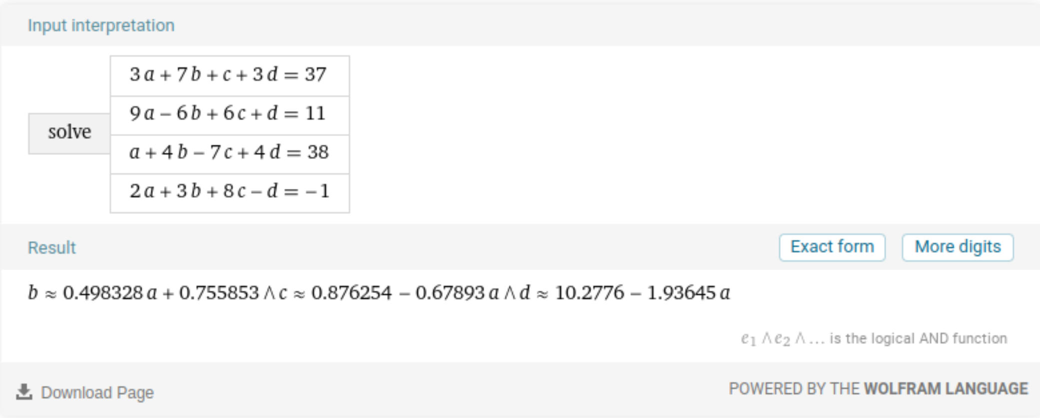

Figure 3: Results from WolframAlpha

1.3 Equation 3

```
A3=np.matrix([[0.0,  1.0,   2.0,   1.0],
              [6.0, -2.0,   3.0,   4.0],
              [0.0,  3.0,   4.0,  -3.0],
              [0.0, -4.0,   3.0,   1.0]])

b3=(np.matrix([2,
               -15,
               10,
               -2])).transpose()
```

We obtain :

$[[-2.]$
$[1.]$
$[1.]$
$[-1.]]$

Verification:

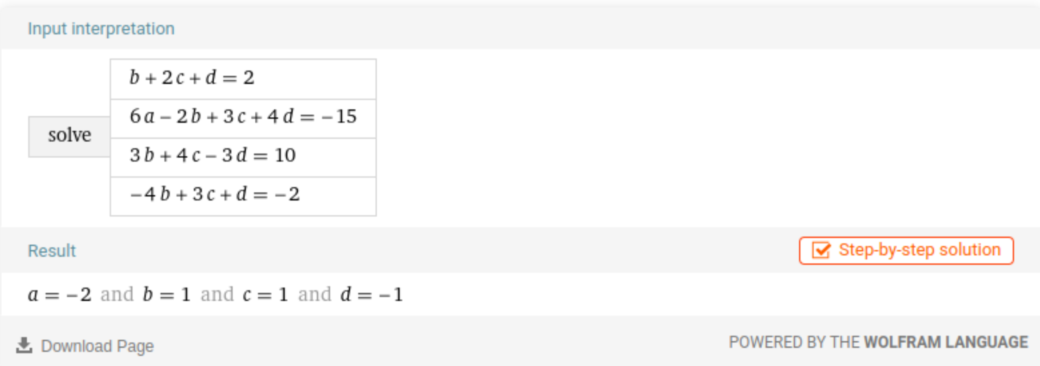

Figure 4: Results from WolframAlpha

1.4 Equation 4

```
A4=np.matrix([[3.0,  7.0,   1.0,   3.0],
              [1.0, -6.0,   6.0,   8.0],
              [4.0,  4.0,  -7.0,   1.0],
              [4.0, 16.0,   2.0,   0.0]])
```

```
b4=(np.matrix([40,
               19,
               36,
               48])).transpose()
```

We obtain :

[[1.]
[3.]
[−2.]
[6.]]

Verification:

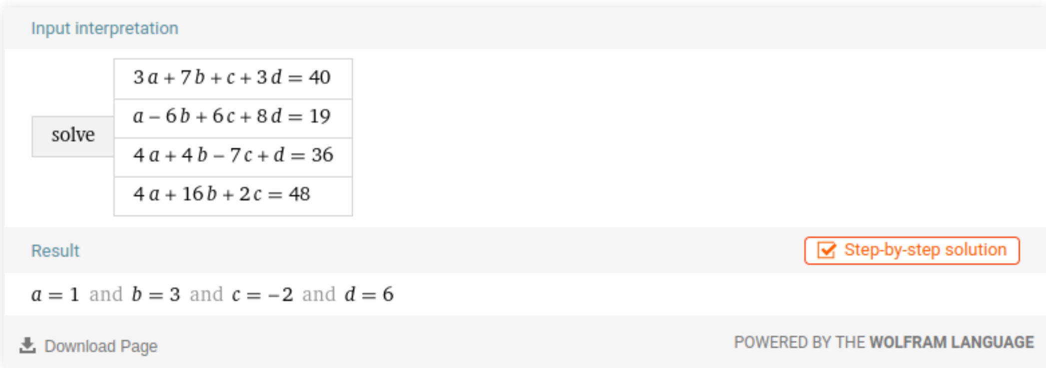

Figure 5: Results from WolframAlpha

1.5 Equation 5

```
A4=np.matrix([[3.0,  7.0,   1.0,   3.0],
              [1.0, -6.0,   6.0,   8.0],
              [4.0,  4.0,  -7.0,   1.0],
              [4.0, 16.0,   2.0,   0.0]])

b5=(np.matrix([-4,
               3,
               7,
               2])).transpose()
```

We obtain no solution for this system :

No solution for this system

Verification:

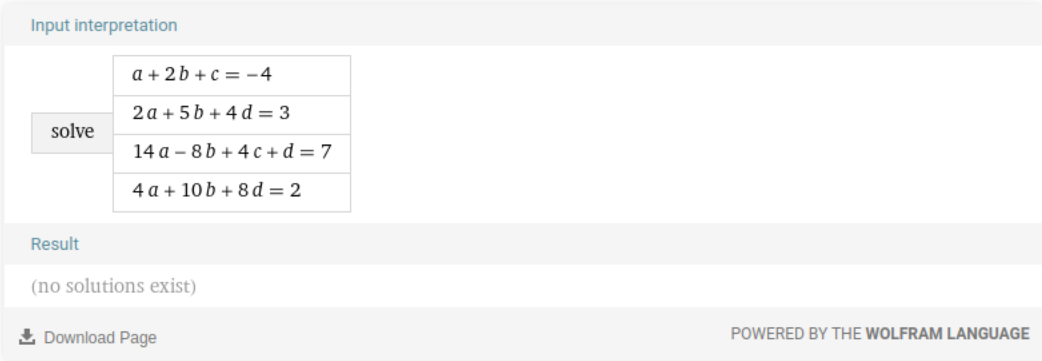

Figure 6: Results from WolframAlpha

2 Non-linear equation system : Quasi-Newton

Then I had to solve the system of non-linear equation below using Quasi-Newton method.

$$\begin{cases} x_2 sin(\frac{x_1}{2}) - 0.1 = 0 & (Z1) \\ x_1^2 + (\frac{x_2}{4})^4 - 12 = 0 & (Z2) \end{cases} \tag{2}$$

First, we can plot the surfaces of the two equations, Z1 and Z2.

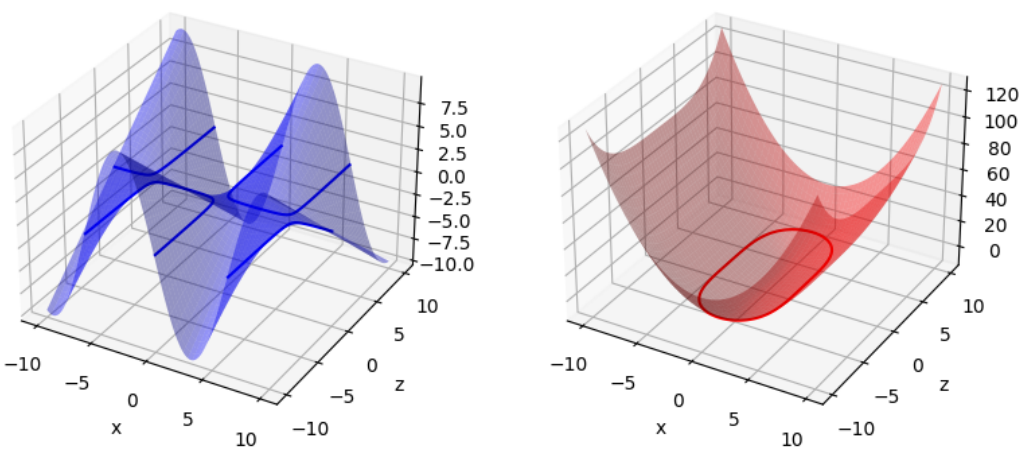

Figure 7: Z1 (blue) and Z2 (red) surfaces

The lines represent the values in which the equation is equal to 0.
Then we can plot the intersections of the two surfaces, and even the intersections of the lines, which will give us the solutions graphically.

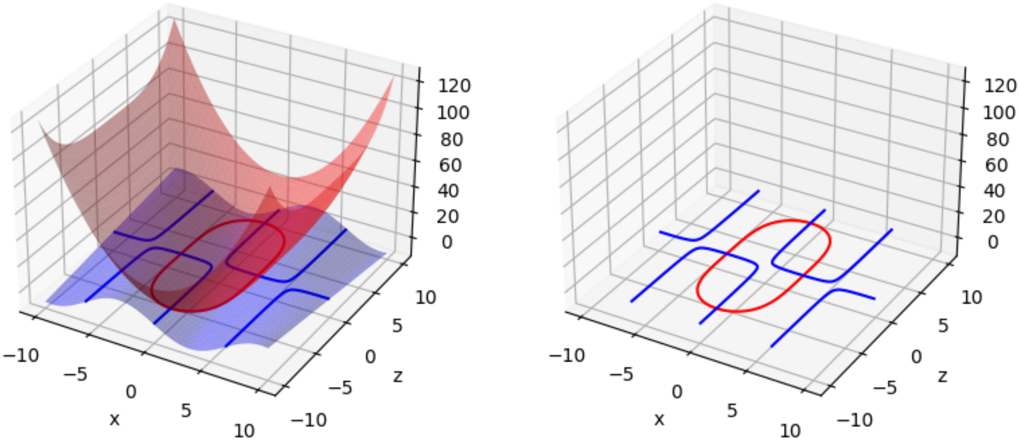

Figure 8: Z1 (blue) and Z2 (red) surfaces intersections

To code the Quasi-Newton algorithm, we will need the function and its Jacobian matrix (obtained with approximations of the partial derivates of the function).

```
def fnleq(x):
    return np.array( [x[1]*np.sin(0.5*x[0])-0.1,
                      x[0]**2+(0.25*x[1])**4-12])

def quasidfi(f, i, x, j):
    h = 1e-6
    xh = []
    for k in range(len(x)):
        if k == i:
            xh.append(x[k] + h)
        else:
            xh.append(x[k])
    y = f(x)[j]
    yh = f(xh)[j]
    r = (yh - y) / h
    #print("r :", r)
    return r


def Jf(fct, x, nbeq):
    multigrad = []
    for f in range(nbeq):
        grad = []
        for i in range(len(x)):
            grad.append(quasidfi(fct, i, x, f))
        multigrad.append(grad)
    return np.array(multigrad)
```

The Quasi-Newton algorithm takes as argument an initial point (with two coordinates in our case), and do several iterations to make this point converge to a root.

```
def QuasiNewton(f, x, eps):
    maxsteps = 2000
    step = 0
    while (step < maxsteps and np.any(abs(f(x)) > eps)):
        step += 1
        #print("xi : ", x)
        #print("s: ", step, " | Jf : ", Jf(f, x, 2))
```

```
        jxtmp = Jf(f, x, 2)
        jxl = jxtmp.tolist()
        if np.abs(jxl[0][0]) < .001 and np.abs(jxl[0][1]) < .001 and np.abs(jxl[1][1]) < .001 and np.abs(jxl[1][0]) < .001 :
            return False, step
        x = np.array(x) - np.linalg.inv(jxtmp)@f(x)
    return x, step
```

Finally, to find all the roots we make a whole set of points (in my case I made a grid of 16 points distributed on our surface of interest (-10 to 10 for the two axis))

```
def createSet(xmin, xmax, ymin, ymax, resolution):
    xy = []
    for yi in range(resolution + 1):
        yivals = []
        yval = ymin + yi * (ymax - ymin) / resolution
        for xi in range(resolution + 1):
            xval = xmin + xi * (xmax - xmin) / resolution
            yivals.append([xval, yval])
        xy.append(yivals)
    return xy

def getAllFromQuasiNewton(f, xy, rooteps):
    sols = []
    totalsteps = 0
    numberofpoints = 0
    # print("Before newton")
    # printcoords(xy)
    for i in range(len(xy)):
        for j in range(len(xy[i])):
            newsol, steps = QuasiNewton(f, xy[i][j], 1e-8)
            totalsteps += steps
            numberofpoints += 1
            if not isinstance(newsol, bool):
                xy[i][j] = newsol.tolist()
                found=False
                for s in sols:
                    if abs(s[0] - newsol[0]) < rooteps and abs(s[1] - newsol[1]) < rooteps:
                        found=True
                if not found:
                    sols.append(newsol.tolist())
                    #print("NEW : ", sols)
    # print("After newton")
    # printcoords(xy)
    print("Average number of steps per point:", totalsteps/numberofpoints)
    return sols


def printcoords(xy):
    print("[")
    for y in xy:
        for x in y:
            print(x, end=", ")
        print("")
    print("]")


xy = createSet(-10, 10, -10, 10, 4)
printPoints(xy)
```

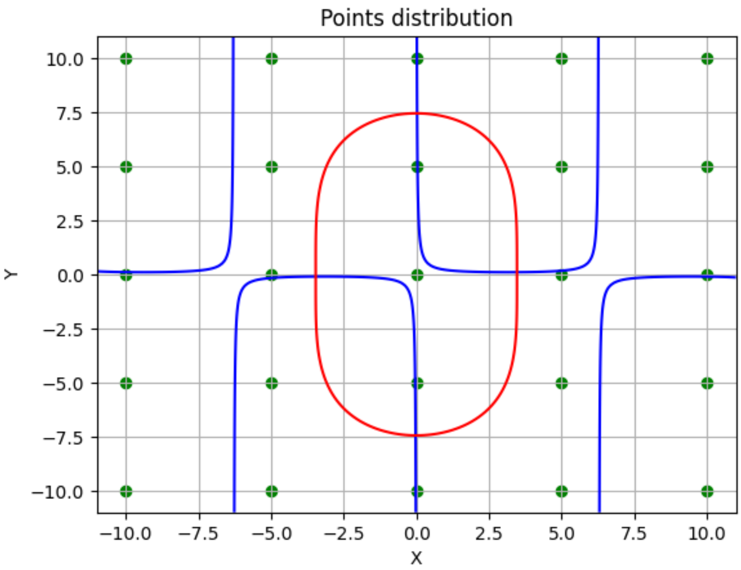

Figure 9: Set of distributed points on which Quasi-Newton will be applied

Then we start Quasi-Newton method on every point and see where they converge.

```
v = getAllFromQuasiNewton(fnleq, xy, 0.1)
print("Solutions of the non-linear equation are : ")
print(v)
printSolutions(v)
```

So finally we obtain the approximated coordinates of the 4 solutions:

Average number of steps per point: 30.64
Solutions of the non-linear equation are :
$[[-0.026865460174149012, -7.4447269258435576],$
$[-3.4641015557326065, -0.10131438702733254],$
$[3.4641015559745827, 0.10131438728799054],$
$[0.026865460174157533, 7.444726925845088]]$

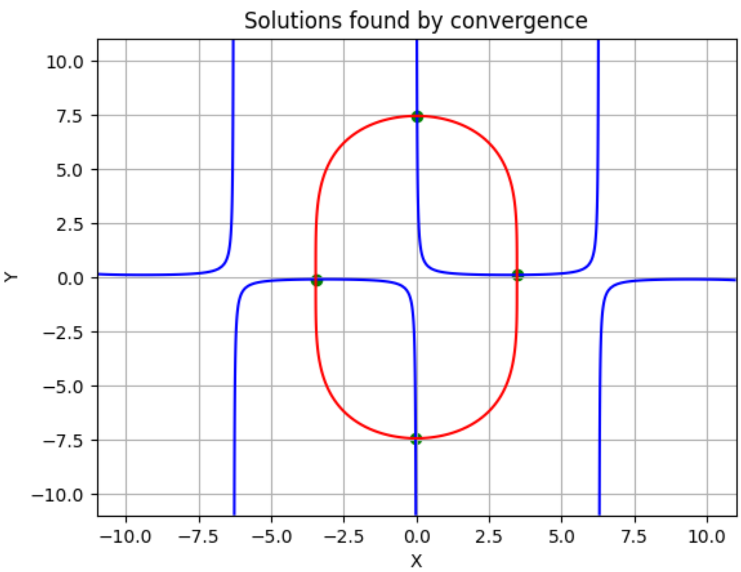

Figure 10: Points convergence after Quasi-Newton method

To verify the results, I used the website `https://www.wolframalpha.com/`.

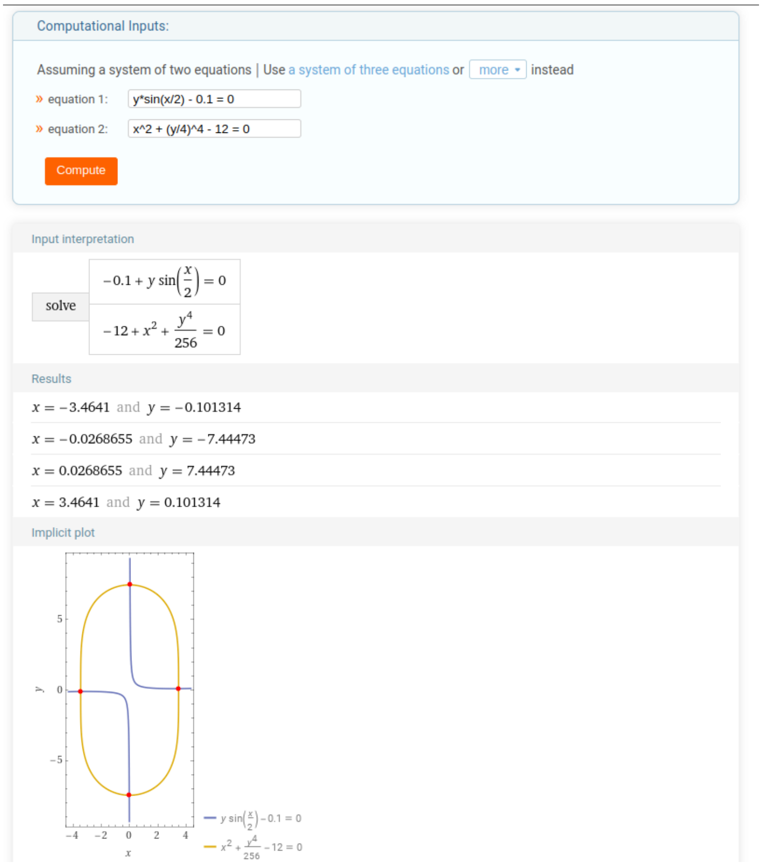

Figure 11: WolframAlpha results

3 Optimization

This problem was solved in group with Enrique Saiz.
The optimization problem was to find the best new places for recycling containers. The problem has to take in account the positions of actual recycling contains as well as geographic data of the city of Kaunas.

3.1 Data used

We get the data from `https://open-data-ls-osp-sdg.hub.arcgis.com/datasets/`, and then we extracted and filtered the data with geopandas python library, and we converted all the geographic coordinates that used EPSG:3346 to common EPSG:4326. We chose to use the population data and relevants economic secotrs in our opinion (primary and secondary sectors, construction, commerce, food sector).

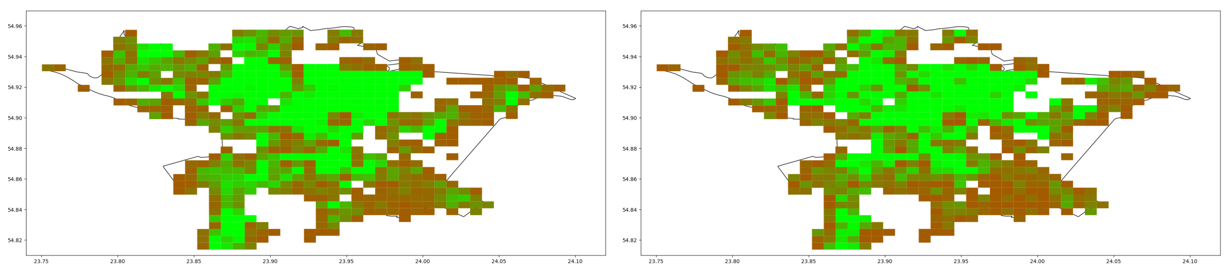

Figure 12: Plots of the population and the economic activity in Kaunas

And then, we normalised and merged the data to get a region of interest.

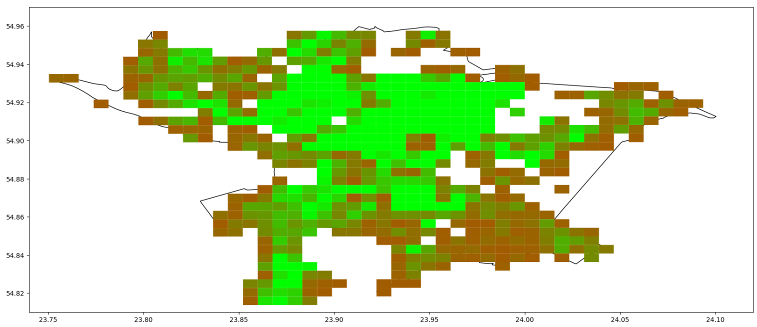

Figure 13: Plot of the normalised region of interest

3.2 Creation of new containers

So, at the beginning we had 567 recycling spots, and we want to add 10 more according to our data :
- Population and relevant sectors in each area
- Distance between points as uniform as possible

So first we spawn 10 random points in our region of interest (according to the positions of the existing containers).

```
# generate and initialize random free positions, to be optimized with gradient
    descent
new = 10

x_values = np.random.uniform(min(positionsFixed[:, 0]), max(positionsFixed[:,
    0]), new)
y_values = np.random.uniform(min(positionsFixed[:, 1]), max(positionsFixed[:,
    1]), new)
positionsFree = np.column_stack((x_values, y_values))

originalPositionsFree = positionsFree.copy()

visualization(positionsFixed, positionsFree)
```

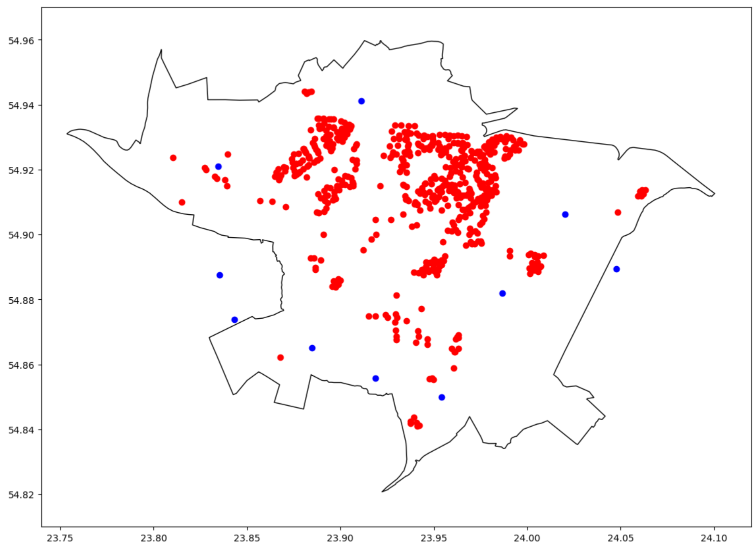

Figure 14: Creation of 10 new points (in blue)

3.3 Optimization of new containers

So to opitmize the positions of the new conntainers, we created an objective function that has its minimum when all the points have the minimal distance between all and are more present in the region of interest. We tried many combinations of the data (multiplications, use of some factors), comparing what had more weight (and what should have more), and we finally found this combination that worked well and give results that seems accurate.

```
def objectiveFunctDistanceImportance(positionsFixed, positionsFree):
    distanceVal = 0
    importanceVal = 0
    avgDist = averageDistanceBetweenAllPoints(positionsFixed, positionsFree)

    # distance value
    for i in range(0, len(positionsFixed)):
        for j in range(0, len(positionsFree)):
            edgeDistance = distanceBetweenTwoPoints(positionsFixed[i],
                positionsFree[j])
            distanceVal += (avgDist - edgeDistance)**2

    for i in range(0, len(positionsFree)):
        for j in range(i+1, len(positionsFree)):
            edgeDistance =  distanceBetweenTwoPoints(positionsFree[i],
                positionsFree[j])
            distanceVal += (avgDist - edgeDistance)**2

    # point value
    for i in range(0, len(positionsFree)):
        importanceVal +=
            1-float(get_normalized_value_at_coordinates(geo_data_merged,
            positionsFree[i, 0], positionsFree[i, 1]))

    return distanceVal + importanceVal
```

To find the minimum of this function, we will need an approximation of the gradient. h is small enough to be a gradient approximation, but big enough to depend on the geographic data (it is 500x500m areas).

```
def quasiGradient(positionsGiven, positionsNew, objectiveFunction, h=0.01):
    f0 = objectiveFunction(positionsGiven, positionsNew)
    df = positionsNew * 0
    for i in range(0, len(positionsNew)):
        for j in range (0, 2): # x and y coordinates
            positionsFreeNew = positionsFree;
            positionsFreeNew[i][j] += h
            f1 = objectiveFunction(positionsGiven, positionsFreeNew) #changed
                from positionsFixed
            df[i][j] = (f1-f0)/h
    return df
```

And finally, we use the gradient descend method to find the minimum of the objective function.

```
def quasiGradientDescent(positionsFixed, positionsFree, objectiveFunction,
    step=0.02, eps=1e-3, maxIter=1000):
    iter = 0
    objValOld = objectiveFunction(positionsFixed, positionsFree)
    objValNew = objValOld
    print("initial: ", objValOld)
    grad = quasiGradient(positionsFixed, positionsFree, objectiveFunction)
    while np.linalg.norm(grad) > eps and iter < maxIter and step > eps:
        grad = grad/np.linalg.norm(grad)
        print("grad", step, grad)
        positionsFree -= step * grad
```

```
        objValNew = objectiveFunction(positionsFixed, positionsFree)
        if objValOld < objValNew:
            positionsFree += step * grad
            step = step * 0.9
        else:
            objValOld = objValNew
        grad = quasiGradient(positionsFixed, positionsFree, objectiveFunction)
        iter += 1
    print("iterations: ", iter,"/",maxIter)
    print("after optimization: ", objValNew)

    objectiveFunction(positionsFixed, positionsFree)
    visualization(positionsFixed, positionsFree, originalPositionsFree, True)
```

To optimize the new points, we just call the function like this.

```
positionsFree = originalPositionsFree.copy()
quasiGradientDescent(positionsFixed, positionsFree,
    objectiveFunctDistanceImportance, maxIter=30)
```

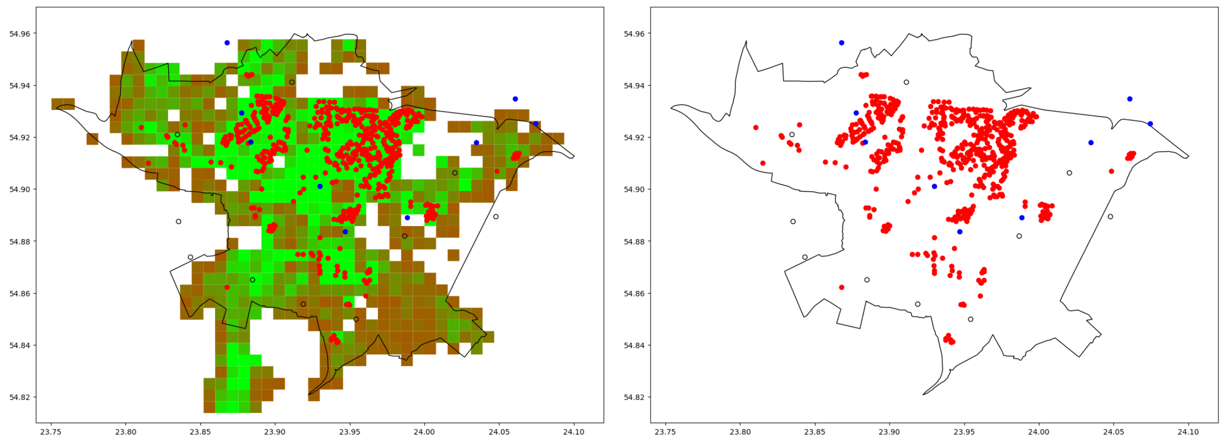

Figure 15: Optimized new points (in blue), with their initial position (black circles)

By comparing the objective function at the points in their original positions and in their new positions, we see that the value decreased as intended.

```
initial: 13.49093762170164
after optimization: 10.698442833207066
```

3.4 Future improvements

Here is a small list of what we could improve on this algorithm in the future:
- Incorporate more features in objective function
- Differentiate the types of current recycling spots
- Repeat computations in high performance environment with 100x100 grid
- Visit and evaluate the new designated spots