HW1

March 3, 2024

[2]:
```
import numpy as np
from sympy import *
from dtumathtools import*
```

1 Problem 1

$q(x_1, x_2, x_3) = x_1^2 + 7x_2^2 + 7x_3^2 + 8x_1x_2 - 8x_1x_3 + 4x_2x_3 + x_1 - 2x_2 + 3$

$q(\underline{x}) = \underline{x}^T * \underline{\underline{A}} * \underline{x} + \underline{x}^T * \underline{b} + c$

It is easy to notice that $\underline{b} = \begin{bmatrix} 1 \\ -2 \\ 0 \end{bmatrix}$ and $c = 3$. The $\underline{\underline{A}}$ matrix is indeed symmetric $\underline{\underline{A}} = \begin{bmatrix} 1 & 4 & -4 \\ 4 & 7 & 2 \\ -4 & 2 & 7 \end{bmatrix}$.
Which can be checked by the following code.

[3]:
```
x1, x2, x3, x4 = symbols('x1, x2, x3, x4')
# defining all needed matrices

x = Matrix([x1, x2, x3])
A = Matrix([[1, 4, -4],[4, 7, 2],[-4, 2, 7]])
b = Matrix([1, -2, 0])
c = Matrix([3])

q = x.T * A * x + x.T * b + c

q.expand()[0]
```

[3]: $x_1^2 + 8x_1x_2 - 8x_1x_3 + x_1 + 7x_2^2 + 4x_2x_3 - 2x_2 + 7x_3^2 + 3$

In orded to achive reduced quadratic form we need to find orthonormal basis $\beta = u_1, u_2, u_3$ for $\underline{\underline{A}}$ such that

$$
\begin{aligned}
(\underline{\underline{U}} * \tilde{\underline{x}})^T * \underline{\underline{A}} * (\underline{\underline{U}} * \tilde{\underline{x}}) + (\underline{\underline{U}} * \tilde{\underline{x}})^T * \underline{b} + c = 0 \\
\tilde{\underline{x}}^T * \underline{\underline{U}}^T * \underline{\underline{A}} * \underline{\underline{U}} * \tilde{\underline{x}} + \tilde{\underline{x}}^T * \underline{\underline{U}}^T * \underline{b} + c = 0 \\
\tilde{\underline{x}}^T * \underline{\underline{D}} * \tilde{\underline{x}} + \tilde{\underline{x}}^T * \underline{\underline{U}}^T * \underline{b} + c = 0,
\end{aligned}
$$

where $\underline{\underline{U}} = \begin{bmatrix} u_1 & u_2 & u_3 \end{bmatrix}$ and $\underline{\underline{D}}$ is a diagonal matrix of corresponding eigenvalues of $\underline{\underline{U}}$.

[5]:
```
#Find the eigenvectors and eigenvalues of A
eigens = A.eigenvects() #returns list of tuples with eigenvalue, its␣
 ↪multiplicity and eigenvectors

#print(eigens)

lamda1 = eigens[0][0] #first eigenvalue
lamda2 = eigens[1][0] #second eigenvalue

u1 = eigens[0][2][0] #first eigenvector lamda1
u2 = eigens[1][2][0] #second eigenvector lamda2
u3 = eigens[1][2][1] #third eigenvector lamda2
print(f"lamda1 = {lamda1}, u1 = {u1} \nlamda2 = {lamda2}, u2 = {u2} u3 = {u3}")
```

```
lamda1 = -3, u1 = Matrix([[2], [-1], [1]])
lamda2 = 9, u2 = Matrix([[1/2], [1], [0]]) u3 = Matrix([[-1/2], [0], [1]])
```

As $\underline{\underline{A}}$ is symmetric then 1) Diagonalization is possible. 2) The different E_λ(eigenvectors belonging to different λ) are orthogonal.

[6]:
```
#Now we can unit vector u1 as it's already orthonormal to u2 and u3
u1 = u1/u1.norm()

#As u2 and u3 come from the same eigenspace we first check the orthogonality
u2.dot(u3) == 0 #Returns False, meaning they are not orthogonal.

#GramSchmidt process can now be applied for u2 and u3 to find an orthonormal␣
 ↪basis.

u2 = u2/u2.norm()
w3 = u3-u3.dot(u2)*u2
u3 = w3/w3.norm()

#print(u1.norm(), u2.norm(), u3.norm())
#print(u1.dot(u2), u2.dot(u3))
```

[7]:
```
#To add vectors u to the matrix U we need to transpose them first and then␣
 ↪transpose the matrix
U = Matrix([u1.T, u2.T, u3.T]).T
#And the diagonal matrix D
D = diag(lamda1, lamda2, lamda2)
#To justify our result we can check if U.T*A*U == D
print(U.T*A*U == D)

U.det() # check if the u1, u2, u3 have the usual orientation
```

```
True
```

[7]:
1

Now we can state the reduced expression for q. **Note** that vector $\underline{x}$ here is given with respect to our new basis.

[9]:
```
q_reduced =  x.T * D * x + x.T * b + c
q_reduced.expand()[0]
```

[9]:
$$-3x_1^2 + x_1 + 9x_2^2 - 2x_2 + 9x_3^2 + 3$$

$$q_{reduced} = -3x_1^2+9x_2^2+9x_3^2+x_1-2x_2+3 = -3(x_1^2-\frac{2}{3}x_1+\frac{1}{9})-x_1+9(x_2^2-\frac{2}{9}x_2+\frac{1}{81})+(3x_3)^2+3+\frac{2}{9} \tag{1}$$

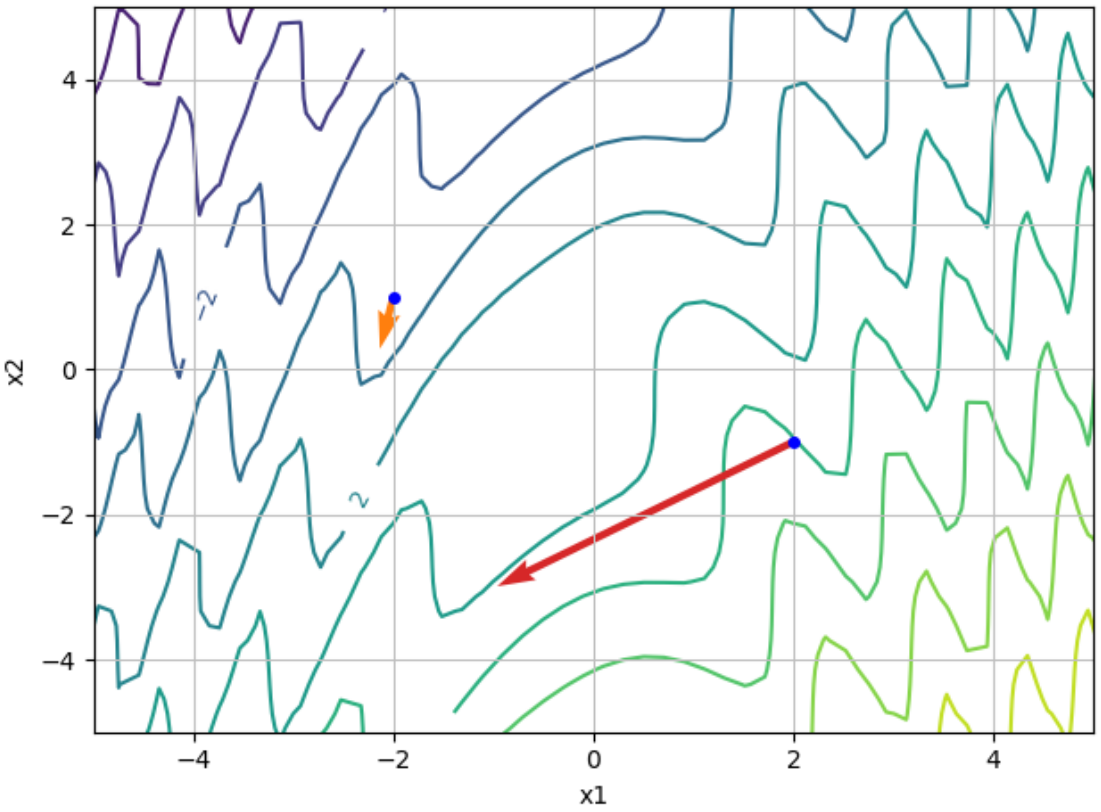
$$q_{reduced} = -3(x_1-\frac{1}{3})^2-x_1+9(x_2-\frac{1}{9})^2+(3x_3)^2+\frac{29}{9} \tag{2}$$

The domain of the function is obviously defined on the whole real axis. As we can see from the expression above, it can take arbitrary positive as well as arbitrary negative values, because of the terms $(-3(x_1-\frac{1}{3})^2-x_1) \to -\infty, x_1 \to +\infty$ and $(9(x_2-\frac{1}{9})^2+(3x_3)^2+\frac{29}{9}) \to +\infty, x_2$ and/or $x_3 \to \pm\infty$.

2 Problem 2

[9]:
```
#defining the f function
def f2(x1,x2):
    return sin(x1**2+x2)+x1-x2+3

#gradient of the function
def nf2(x1_val,x2_val):
    return dtutools.gradient(f2(x1, x2)).subs({x1: x1_val, x2: x2_val}).evalf()

#two points of the gradient evaluation
x0 = np.array([-2,1])
x0_2 = np.array([2,-1])

#ploting the level curves with plot_contour function and adding gradient␣
 ↪vectors to the plot
lvl = dtuplot.plot_contour(f2, (-6, 6), (-6, 6), rendering_kw = {"levels":␣
 ↪[*range(-8,15,2)]}, xlabel = "x1", ylabel = "x2", is_filled=False,␣
 ↪show=False)
lvl.extend(dtuplot.scatter(x0, rendering_kw={"markersize":4,"color":'b'},␣
 ↪show=False))
lvl.extend(dtuplot.quiver(x0,nf2(*x0),show=False))
lvl.extend(dtuplot.scatter(x0_2, rendering_kw={"markersize":4,"color":'b'},␣
 ↪show=False))
lvl.extend(dtuplot.quiver(x0_2,nf2(*x0_2),show=False))
lvl.xlim = [-5,5]
lvl.ylim = [-5,5]
```

```
lvl.show()
```



The projection of the gradient vector defined by the $nf2(-2,1)$ function on the positive x_1 direction is negative, which means it points backward to the positive x_1 direction. It can be seen from the level curves that the function in that direction is decreasing. The gradient at the point $(2,-1)$ is also decreasing for the same reasons.

3 Problem 3

[6]:
```
y1, y2, y3 = symbols('y1, y2, y3')
#f function
def f3(x1_v,x2_v,x3_v,x4_v):
    return Matrix([x2*x3 + x1**2, -x1*x4 + x2**3, x1*x2*x3*x4]).subs({x1: x1_v,␣
 ↪x2: x2_v, x3: x3_v, x4: x4_v})

#Jacobian matrix of a g function
def Jg(y1_v, y2_v, y3_v):
    return Matrix([1+y3+2*y1*y2, y1**2+2*y2*y3**2, 3+y1+2*y3*y2**2]).subs({y1:␣
 ↪y1_v, y2: y2_v, y3: y3_v})
```

```
#Jacobian matrix of the f function
def Jf3(x1_v,x2_v,x3_v,x4_v):
    return f3(x1,x2,x3,x4).jacobian([x1,x2,x3,x4]).subs({x1: x1_v, x2: x2_v, x3:
 ↪ x3_v, x4: x4_v})

Jf3(x1,x2,x3,x4), Jg(y1, y2, y3)
```

[6]:
```
(Matrix([
 [    2*x1,       x3,       x2,        0],
 [     -x4,  3*x2**2,        0,      -x1],
 [x2*x3*x4, x1*x3*x4, x1*x2*x4, x1*x2*x3]]),
 Matrix([
 [   2*y1*y2 + y3 + 1],
 [ y1**2 + 2*y2*y3**2],
 [y1 + 2*y2**2*y3 + 3]]))
```

[9]:
```
Jf3(1,-2,-1,4) #evaluation
```

[9]:
$$\begin{bmatrix} 2 & -1 & -2 & 0 \\ -4 & 12 & 0 & -1 \\ 8 & -4 & -8 & 2 \end{bmatrix}$$

Using Theorem 3.8.4 (Generalized chain rule) the Jacobian matrix fot the function $g \circ f$ can be found as follows:

[8]:
```
def Jgf(x1_v,x2_v,x3_v,x4_v):
    A = Jf3(x1_v,x2_v,x3_v,x4_v)
    B = Jg(*f3(x1_v,x2_v,x3_v,x4_v)).T
    return B*A
#Jgf(x1,x2,x3,x4)
Jgf(1,-2,-1,4) #evaluation at the point(1,-2,-1,4)
```

[8]:
$$\begin{bmatrix} 24462 & -27501 & -18354 & 6147 \end{bmatrix}$$

4 Problem 4

The idea of the program below is to take an arbitrary 4-dimensional complex vector and consequently find 3 orthogonal vectors to the given one. This is being done by solving a system of equations like

$$< \underline{v}, \underline{v}_1 >= 0$$

$$\begin{cases} < \underline{v}_2, \underline{v}_1 >= 0 \\ < \underline{v}_2, \underline{v} >= 0 \end{cases}$$

$$\begin{cases} < \underline{v}_3, \underline{v}_2 >= 0 \\ < \underline{v}_3, \underline{v}_1 >= 0 \\ < \underline{v}_3, \underline{v} >= 0 \end{cases}$$

, $where\, v, v_1, v_2, v_3 \in \mathbb{C}$

However, the problem with this method is that we have more unknown parameters than equations, for example:

$$\begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix} = \begin{bmatrix} k * x2 + t * x3 + c * x4 \\ x2 \\ x3 \\ x4 \end{bmatrix}, where\, k, t, c \in \mathbb{C}$$

Therefore x_2, x_3 and x_4 are randomly chosen in the range of integers $[-2, 2]$. Obviously having more equations in a system yields fewer unknowns, therefore for the first vector, we peak 3 random values for x_1, x_2, x_3, for the second vector - 2 random values for x_3, x_4 and for the fourth only one for x_4.

The last step will be the normalization of each of them.

[12]:
```
#To be able to find an inner product of two complex vectors it's convenient to␣
 ↪define a new function
def cdot(x1, x2):
    return x1.dot(x2, conjugate_convention = 'right')
```

[15]:
```
v=Matrix([1-I,2-I,I,I]) #define a beginning vector

# cbasis function takes a 4-dimensional vector as an argument and returns 4␣
 ↪vectors that span an orthonormal basis
# with the first vector being normalized given one.

def cbasis(v):
    v1=Matrix([x1,x2,x3,x4])
    eq1 = cdot(v1,v) #defining the first equation for the inner product
    sol = solve((eq1,0),x1,x2,x3,x4) #solving that equation
    v1 = v1.subs(sol) #substitution of the solution

    # Substitution random parameters and checking if the vector's norm does not␣
 ↪equal zero.
    while True:
        sub = np.random.default_rng().integers(-2, 2, size=3) #generating 3␣
 ↪random integers in the range [-2, 2]
        k = v1.subs({x2: sub[0], x3:sub[1], x4:sub[2]}) #substitution these␣
 ↪parameters in the vector
        if k.norm() != 0: #check the norm value
            v1 = k
            break

    #THE SAME PROCEDURE IS PERFORMED 2 MORE TIMES BUT INCREASING THE NUMBER OF␣
 ↪EQUATIONS FOR THE INNER PRODUCT BETWEEN VECTORS

    v2 = Matrix([x1,x2,x3,x4])
```

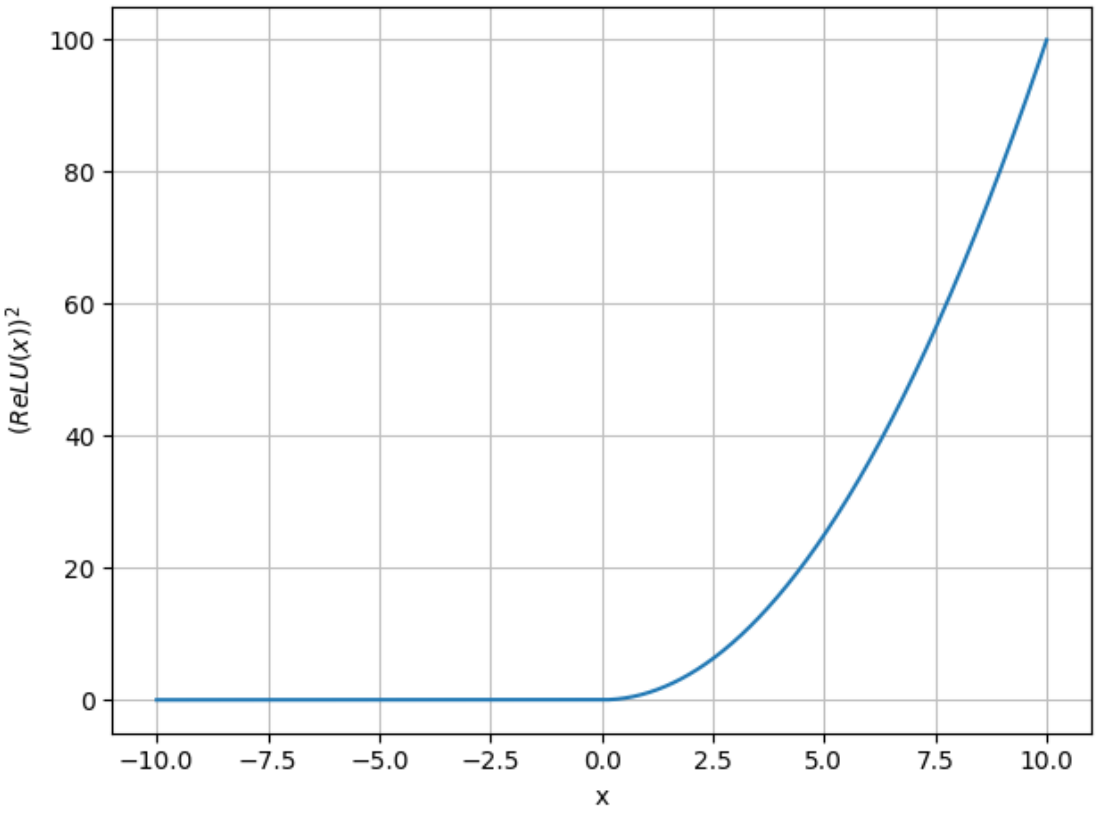
```
    eq1 = Eq(cdot(v2,v1),0)
    eq2 = Eq(cdot(v2,v),0)
    sol = solve([eq1, eq2], x1,x2,x3,x4)
    v2 = v2.subs(sol)
    while True:
        sub = np.random.default_rng().integers(-2, 2, size=2)
        k = v2.subs({x3: sub[0], x4: sub[1]})
        if k.norm() != 0:
            v2 = k
            break

    v3 = Matrix([x1,x2,x3,x4])
    eq1 = Eq(cdot(v3,v1),0)
    eq2 = Eq(cdot(v3,v2),0)
    eq3 = Eq(cdot(v3,v), 0)
    sol = solve([eq1, eq2, eq3], x1,x2,x3,x4)
    v3 = v3.subs(sol)
    while True:
        sub = np.random.default_rng().integers(-2, 2, size=1)
        k = v3.subs({x4: sub[0]})
        if k.norm() != 0:
            v3 = k
            break

    v= v/v.norm()
    v1 = v1/v1.norm()
    v2 = v2/v2.norm()
    v3 = v3/v3.norm()
    return v, v1, v2, v3

u1, u2, u3, u4 = cbasis(v)

u1, u2, u3, u4

# The inner product that can be evaluated as follows below would not␣
 ↪necessarily give exactly 0,
# more likely something like -0.e-127 - 0.e-127*I: because of the python
# computations, however, if you were to compute the values by hand it would␣
 ↪yield 0.

#cdot(u3, u1).evalf(), cdot(u3, u2).evalf(), cdot(u3, u4).evalf()
```

[15]:
```
(Matrix([
 [1/3 - I/3],
 [2/3 - I/3],
 [      I/3],
 [      I/3]]),
```

```
Matrix([
[sqrt(6)*(1/2 + I/2)/3],
[                    0],
[                    0],
[           sqrt(6)/3]]),
Matrix([
[                        0],
[sqrt(30)*(-1/5 - 2*I/5)/6],
[              -sqrt(30)/6],
[                        0]]),
Matrix([
[ sqrt(2)*(-1 - I)/3],
[sqrt(2)*(1/2 + I)/3],
[         -sqrt(2)/6],
[          sqrt(2)/3]]))
```

5 Problem 5

[4]:
```
#defining (ReLU(x))^2 function
def relu(x):
    return (np.maximum(0,x))**2
dtuplot.plot(relu, (-10,10), xlabel='x', ylabel=r'$(ReLU(x))^2$')
```



[4]: <spb.backends.matplotlib.matplotlib.MatplotlibBackend at 0x23b515200d0>

The function $f(x) = (ReLU(x))^2$ is differentiable on the entire real axis as x^2 is definately differentiable on the $x > 0$, when $x < 0$ the $f^{'}(x) = 0$ and for the most interesting point $x = 0$: as x approaches 0 from the right the derivative of the function is $2x = 0$ and when it approaches from the left the derivative is again equals to 0, which indicates that slope of the function at that point is the same whether we approach from the right or the left, hence derivative exists.

Contrary to $f(x)$ the derivative $f^{'}(x)$ is not differentiable on the entire real axis as if we take a look at the point $x = 0$ the derivative when approaching from the right equals $(2x)^{'} = 2$, while when we approach it from the left the derivative equals to 0, hence, it is not differentable at this point.

To prove the point that $f^{'}(x)$ is continuous on the entire real axis we can use the next definition of continuity:

$$\forall \epsilon > 0 \exists \delta > 0 \forall x \in I : |x - x_0| < \delta \rightarrow |f(x) - f(x_0)| < \epsilon$$

It is easy to begin with the left part of the function where it is a constant, $x < 0$. As $f^{'}(x) = f^{'}(x_0)$ the $|f^{'}(x) - f^{'}(x_0)| < \epsilon$ works out automatically for any ϵ, and δ can arbitrarily be chosen to fit the condition $|x - x_0| < \delta$. For the part where $x \geq 0$:

$$\begin{aligned}
&f^{'}(x) = 2x,\ x \geq 0 \\
&|x - x_0| < \delta \rightarrow |f^{'}(x) - f^{'}(x_0)| < \epsilon \\
&|2x - 2x_0| < \delta \\
&2|x - x_0| < \delta \\
&\text{Let } \delta = \frac{\epsilon}{2} \\
&|x - x_0| < \frac{\epsilon}{2} \Rightarrow 2|x - x_0| < \epsilon \\
&|f^{'}(x) - f^{'}(x_0)| = 2|x - x_0| < \epsilon
\end{aligned}$$

So $f^{'}$ is continuous.