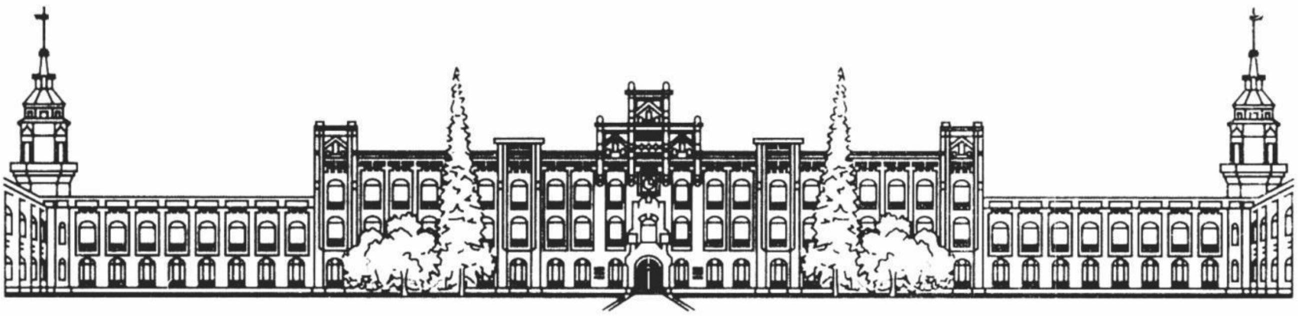


Міністерство освіти і науки України Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Інститут прикладного системного аналізу

Лабораторна робота №3

з курсу «Чисельні методи»
з теми «Методи розв'язання нелінійних систем»

Варіант №4

Виконав: студент 2 курсу групи КА-02
Козак Назар Ігорович
Перевірила: старший викладач
Хоменко Ольга Володимирівна

Київ-2022

Мета роботи: навчитися застосовувати чисельні методи розв'язання нелінійних систем.

Хід роботи:

Завдання 1

1. Розв'язати систему 1 методом простих ітерацій. Для цього:
    - визначити початкове наближення, побудувавши графіки кривих системи;
    - перевірити достатні умови збіжності з детальним поясненням (задати область, в якій перевірити виконання умов збіжності, можна робити фото написаного і вставляти в звіт);
    - реалізувати метод простих ітерацій. Розв'язати систему з точністю **$\varepsilon = 10^{-5}$**;
    - програмний код надіслати в класрум в окремому файлі та вставити текст програми у звіт.
2. Результати роботи програми оформити у звіті у вигляді таблиці. Якщо ітерацій більше 15, в таблицю записати лише перші 15.

№ ітерації			Δ
0			
1			
…			

3. Виконати перевірку, обчисливши F(x_*)
4. Задати декілька інших початкових наближень (які не близькі до розв'язку) та з'ясувати як змінюється при цьому ітераційний процес, написати про це у висновку.
5. Знайти розв'язок системи за допомогою fsolve бібліотеки scipy.optimize

Завдання 2

1. Розв'язати систему 2 методом Ньютона (або спрощеним методом Ньютона). Для цього

    - визначити початкове наближення, побудувавши графіки кривих системи;

- реалізувати метод Ньютона (або спрощений метод Ньютона). За потреби можна використовувати функції linalg.solve та ін. Розв’язати систему з точністю **ε= 10^{-5}**;
- програмний код надіслати в класрум в окремому файлі та вставити текст програми у звіт.

2. Результати роботи програми оформити у звіті у вигляді таблиці. Якщо ітерацій більше 15, в таблицю записати лише перші 15.

№ ітерації			Δ
0			
1			
…			

3. Виконати перевірку, обчисливши F(x_*)

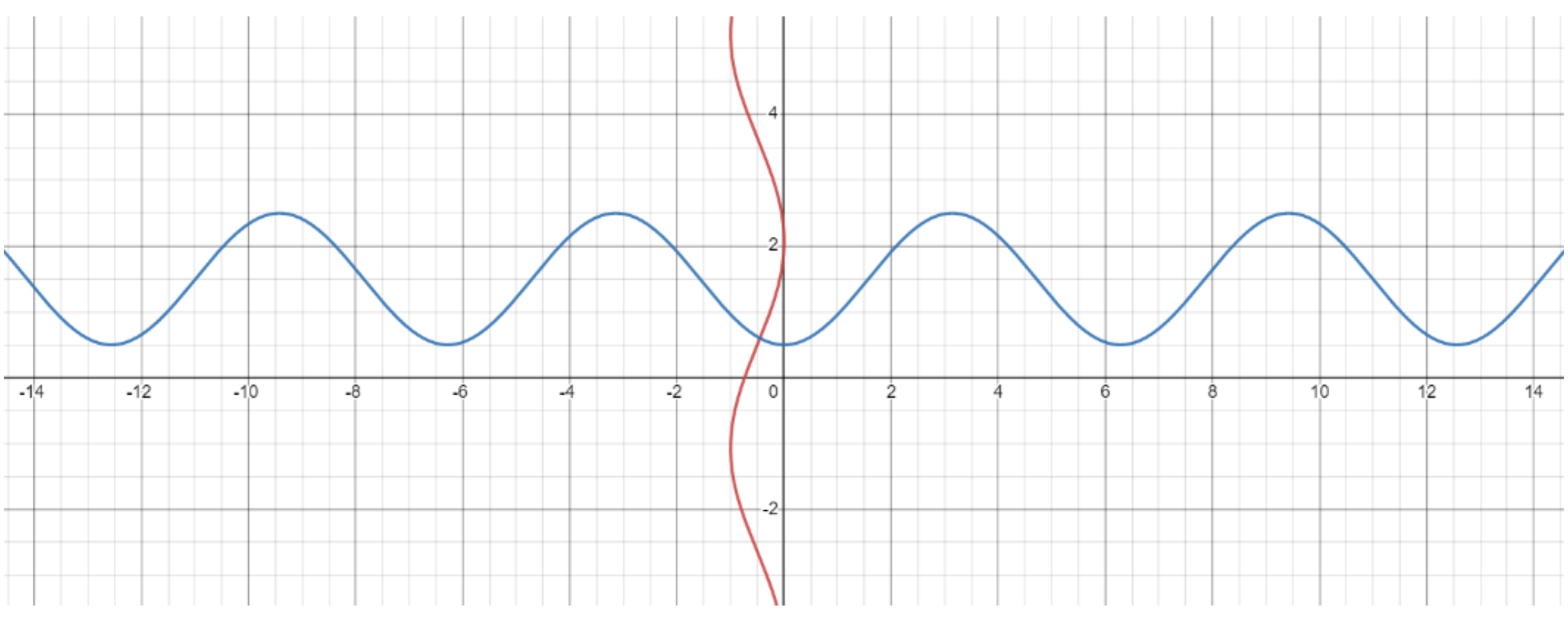
4. Задати декілька інших початкових наближень (які не близькі до розв’язку) та з’ясувати як змінюється при цьому ітераційний процес, написати про це у висновку.

5. Знайти розв’язок системи за допомогою fsolve бібліотеки scipy.optimize

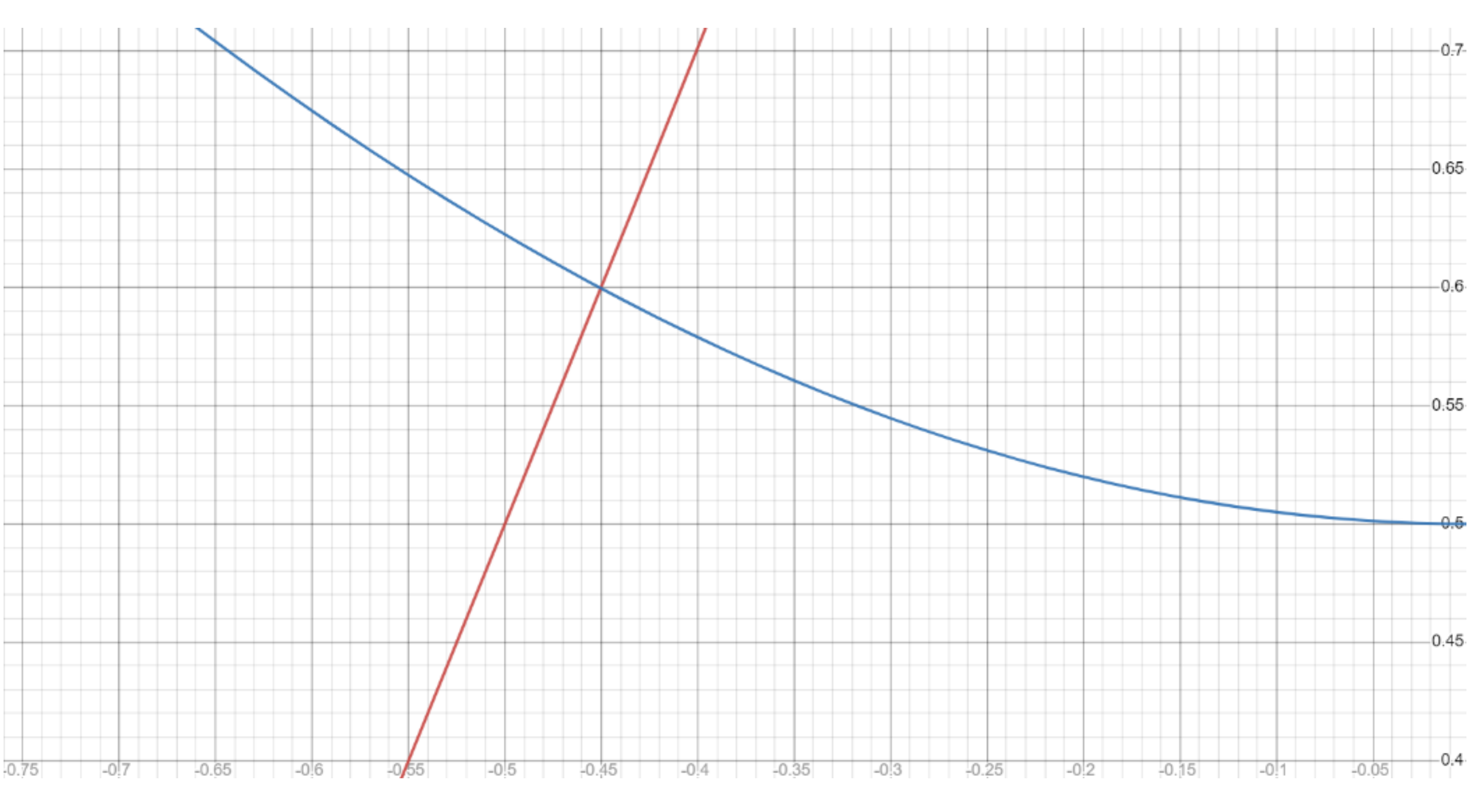
Завдання. Номер № варіанту співпадає з номером завдання. 1) розв’язуємо методом простих ітерацій, 2) – методом Ньютона або спрощеним методом Ньютона.

Завдання 1

Розв'язати систему $\begin{cases} \cos x + y = 1.5 \\ 2x - \sin(y - 0.5) = 1 \end{cases}$ методом простих ітерацій



і трошки ближче:



$x - \frac{1}{2}\sin(y - 0.5) - \frac{1}{2} = 0$ − позначено червоним

$y - 1.5 + \cos x = 0$ − позначено синім

За початкове наближення візьмемо точку $x^{(0)} = (-0.4; 0.6)$

Перевірити достатню умову збіжності

Покладемо $\varphi_1(x, y) = \frac{1}{2}\sin(y - 0.5) + \frac{1}{2}$; $\varphi_2(x, y) = 1.5 - \cos x$

Достатня умова збіжності для метода простих ітерацій має вигляд:

$$\max_{x \in G} \max_{i} \sum_{j=1}^{n} \left| \frac{\partial \varphi_i(x)}{\partial x_j} \right| \le q < 1$$

або

$$\max_{x \in G} \max_{j} \sum_{i=1}^{n} \left| \frac{\partial \varphi_i(x)}{\partial x_j} \right| \le q < 1$$

Будемо перевіряти першу умову. Виберемо такий окіл точки:

$$x^{(0)} : G = \{(x, y) : |x + 0.4| \le 0.1;\ |y - 0.6| \le 0.1\}$$

Знайдемо частинні похідні

$$\frac{\partial \varphi_1}{\partial x} = \frac{\partial}{\partial x}\left(\frac{1}{2}\sin(y - 0.5) + \frac{1}{2}\right) = 0$$

$$\frac{\partial \varphi_1}{\partial y} = \frac{\partial}{\partial y}\left(\frac{1}{2}\sin(y - 0.5) + \frac{1}{2}\right) = \frac{1}{2}\cos(y - 0.5)$$

$$\frac{\partial \varphi_2}{\partial x} = \frac{\partial}{\partial x}(1.5 - \cos x) = \sin x$$

$$\frac{\partial \varphi_2}{\partial y} = \frac{\partial}{\partial y}(1.5 - \cos x) = 0$$

Оцінимо їх модулі в області G:

$$\left| \frac{\partial \varphi_1}{\partial y} \right| = \frac{1}{2}|\cos(y - 0.5)|$$

Ця функція залежить лише від y. Побудуємо її графік на проміжку [0.5, 0.7], щоб побачити в якій точці функція набуває найбільшого значення

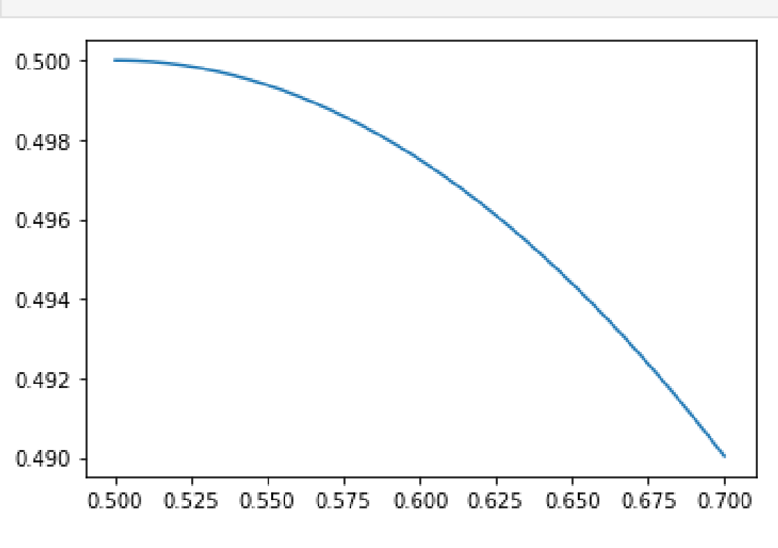
In [11]:

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

def f_1(y):
    return (np.absolute(np.cos(y-0.5)))/2

fig = plt.subplots()
y = np.linspace(0.5,0.7,100)
plt.plot(y, f_1(y))

plt.show()
```



Отже, робимо висновок, що $\max_{(x,y) \in G} \left| \frac{\partial \varphi_1}{\partial y}(x, y) \right| = 0.5$, тому $\left| \frac{\partial \varphi_1}{\partial y} \right| \le 0.5$

Аналогічно зробимо для $\left| \frac{\partial \varphi_2}{\partial x} \right| = |\sin x|$. Ця функція залежить лише від x, який в області G лежить у проміжку $[-0.5, -0.3]$

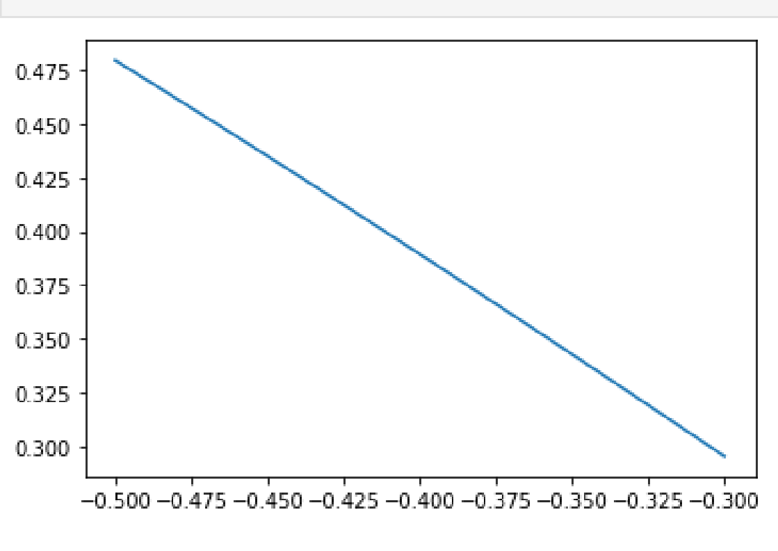
In [12]:

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

def f_2(x):
    return np.absolute(np.sin(x))

fig = plt.subplots()
x = np.linspace(-0.3,-0.5,100)
plt.plot(x, f_2(x))

plt.show()
```



Отже, максимального значення $\left| \frac{\partial \varphi_2}{\partial x} \right|$ приймає коли $x = -0.5$. Обчислимо це значення:

In [13]:

```
np.absolute(np.sin(-0.5))
```

Out[13]:

```
0.479425538604203
```

Отримали, що $\max_{(x,y) \in G} \left| \frac{\partial \varphi_2}{\partial x}(x, y) \right| = |\sin(-0.5)| < 0.5$

Тепер можна перевірити власне умову збіжності:

$$\left| \frac{\partial \varphi_1}{\partial x}(x, y) \right| + \left| \frac{\partial \varphi_1}{\partial y}(x, y) \right| = 0.5 < 1$$

$$\left| \frac{\partial \varphi_2}{\partial x}(x, y) \right| + \left| \frac{\partial \varphi_2}{\partial y}(x, y) \right| = |\sin(-0.5)| < 0.5 < 1$$

Отже, умова збіжності - виконується

Реалізувати метод простих ітерацій. Розв'язати систему з точністю $\varepsilon = 10^{-5}$

In [14]:

```
import math

def delta(x_k, x_k_plus_1):
    return max(abs(x_k_plus_1[0]-x_k[0]), abs(x_k_plus_1[1]-x_k[1]))

def simple_iter(phi, x_0):

    x_1 = phi(x_0)
    data = np.vstack((np.array([[x_0[0], x_0[1]]]), np.array([[x_1[0], x_1[1]]])))
    deltas = np.vstack((np.array([[0]]), np.array([[delta(x_0, x_1)]])))


    while delta(x_0, x_1) > 0.00001:

        x_0 = x_1
        x_1 = phi(x_0)

        data = np.vstack((data, np.array([[x_1[0], x_1[1]]])))
        deltas = np.vstack((deltas, np.array([[delta(x_0,x_1)]])))

    iter_info = np.hstack((data, deltas))

    return x_1, iter_info
```

In [15]:

```
def phi(x):
    return ((1/2)*np.sin(x[1]-0.5) +(1/2), 1.5 - np.cos(x[0]))

x_0 = (-0.4, 0.6)
Result, iter_info = simple_iter(phi, x_0)

Result
```

Out[15]:

```
(0.5819217672655836, 0.6645926439465167)
```

In [16]:

```
# подивимось скільки ітерацій було зроблено
iter_info.shape[0]
```

Out[16]:

```
17
```

Результати роботи програми оформити у вигляді таблиці.

Результати роботи програми записані в масив data. Перетворимо цей масив в об'єкт типу pandas.DataFrame, щоб його можна було вивести у вигляді таблиці. Для цього створимо окрему функцію.

In [61]:

```
import pandas as pd

def to_df(array):

    #виведемо перші 15 ітерацій.
    array = array[:15]

    num = array.shape[0]
    df = pd.DataFrame(array, columns = [ 'x','y','delta'], index = np.arange(1,num+1))

    # це поле нічому не відповідає
    df['delta'][0] = None


    return df
```

In [62]:

```
to_df(iter_info)
```

Out[62]:

	x	y	delta
1	-0.400000	0.600000	0.000000
2	0.549917	0.578939	0.949917
3	0.539429	0.647432	0.068493
4	0.573449	0.641998	0.034021
5	0.570760	0.659965	0.017968
6	0.579642	0.658510	0.008882
7	0.578923	0.663341	0.004832
8	0.581308	0.662948	0.002385
9	0.581114	0.664255	0.001307
10	0.581759	0.664148	0.000645
11	0.581706	0.664502	0.000354
12	0.581881	0.664474	0.000175
13	0.581866	0.664570	0.000096
14	0.581914	0.664562	0.000047
15	0.581910	0.664588	0.000026

Виконати перевірку, обчисливши $F(x\_^*)$

In [19]:

```
def F(x):
    return ((1/2)*np.sin(x[1]-0.5) +(1/2) - x[0], 1.5 - np.cos(x[0]) - x[1])

F(Result)
```

Out[19]:

```
(3.4796770898015694e-06, -5.754404523994339e-07)
```

Задати декілька інших початкових наближень (які не близькі до розв'язку) та з'ясувати як змінюється при цьому ітераційний процес, написати про це у висновку.

Візьмемо п'ять випадкових наближень і створимо табличку, щоб можна було побачити чи змінюється при цьому ітераційний процес.

In [20]:

```
# np.random.rand повертає матрицю заповнену випадковими числами в проміжку від 0 до 1.
# Тому щоб отримати від'ємні числа, а також числа по модулю більші одиниці зробимо певні пертворення:

initial_approximations = (np.random.rand(5,2)-0.5)*100

new_data = np.zeros((5,5))

for i in range(5):

    new_data[i][0] = initial_approximations[i][0]
    new_data[i][1] = initial_approximations[i][1]

    result, info  = simple_iter(phi, initial_approximations[i])

    new_data[i][2] = info.shape[0]
    new_data[i][3] = result[0]
    new_data[i][4] = result[1]

pd.DataFrame(new_data, columns = [ 'x_0','y_0','iter', 'x', 'y'], index = np.arange(1,6))
```

Out[20]:

	x_0	y_0	iter	x	y
1	-39.186847	0.338142	21.0	0.581929	0.664594
2	-0.930339	21.385439	20.0	0.581930	0.664597
3	-18.161740	-33.466778	19.0	0.581927	0.664591
4	-26.731528	-28.055202	21.0	0.581930	0.664595
5	6.231818	-16.775199	21.0	0.581926	0.664597

Тут iter це кількість ітерацій, x_0, y_0 відповідно перша та друга координата початкового наближення, а x,y відповідно перша та друга координата розв'язку. Як бачимо точність майже не міняється, а кількість операцій хоч і збільшується, але не на багато.

Знайти розв'язок системи за допомогою fsolve бібліотеки scipy.optimize

In [21]:

```
from scipy.optimize import fsolve

x_0 = (-0.4, 0.6)
print("розв'язок отриманий за допомогою fsolve:", tuple(fsolve(F, x_0)))
print("розв'язок отриманий за допомогою МПІ    :", Result)
```

```
розв'язок отриманий за допомогою fsolve: (0.5819261517549239, 0.6645944783630967)
розв'язок отриманий за допомогою МПІ    : (0.5819217672655836, 0.6645926439465167)
```

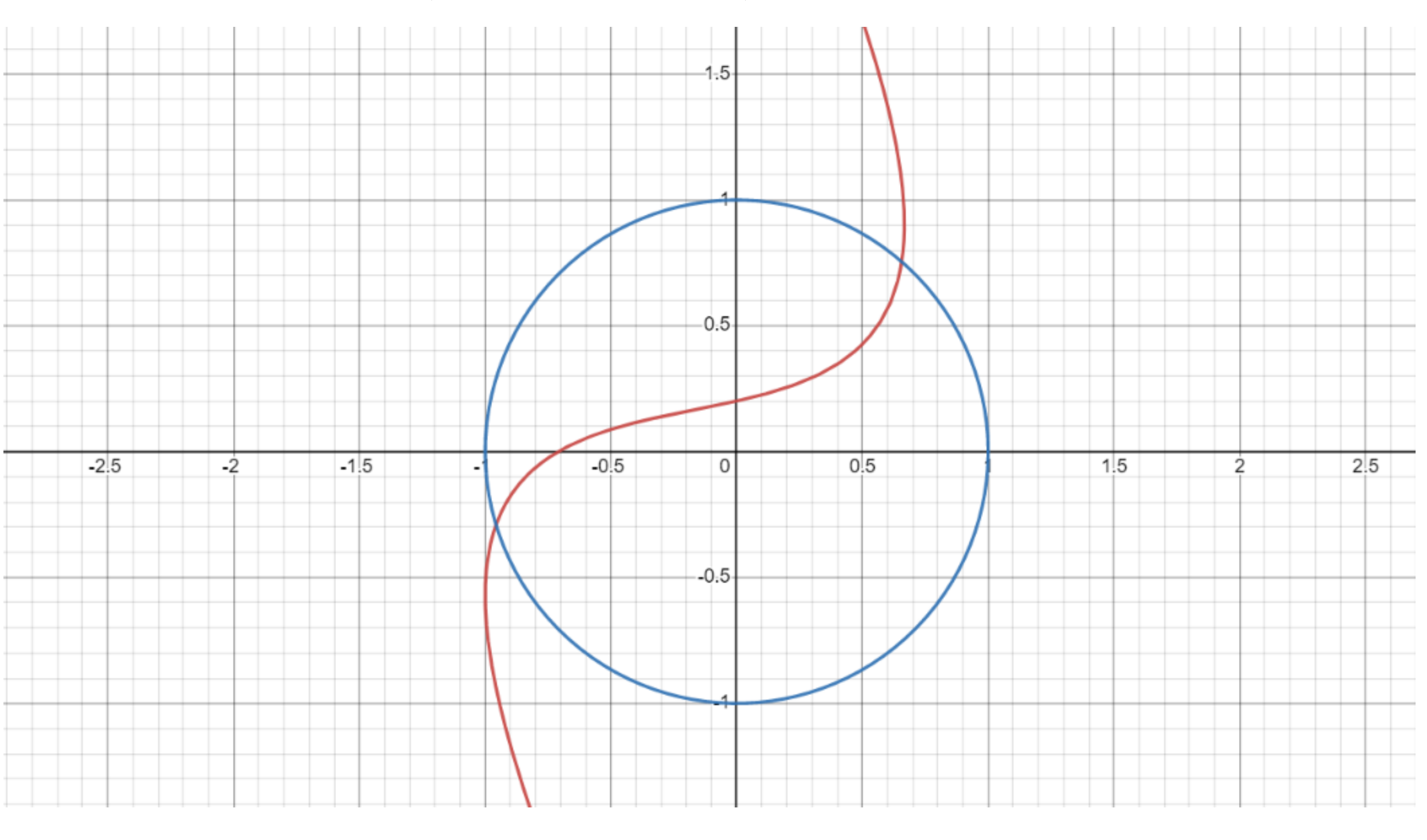
*МПІ - метод простих ітерацій

Завдання 2

Розв'язати систему $\begin{cases} \sin(x + y) - 1.2x = 0.2 \\ x^2 + y^2 = 1 \end{cases}$ методом Ньютона(або спрощеним методом Ньютона).

Визначити початкове наближення, побудувавши графіки кривих системи

$$\begin{cases} \sin(x + y) - 1.2x = 0.2 \\ x^2 + y^2 = 1 \end{cases} \Rightarrow \begin{cases} \sin(x + y) - 1.2x - 0.2 = 0 \\ x^2 + y^2 - 1 = 0 \end{cases}$$



Будемо шукати корінь, що знаходиться в третьому квадранті. Для нього за початкове наближення візьмемо точку $x^{(0)} = (-1, -0.25)$

Реалізувати метод Ньютона(або спрощений метод Ньютона)

Покладемо $f_1(x, y) = \sin(x + y) - 1.2x - 0.2$; $f_2(x, y) = x^2 + y^2 - 1$. Знайдемо матрицю Якобі вектор функції $\vec{F} = (f_1, f_2)^T$

$$W(x, y) = \begin{pmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{pmatrix} = \begin{pmatrix} \frac{\partial}{\partial x}(\sin(x + y) - 1.2x - 0.2) & \frac{\partial}{\partial y}(\sin(x + y) - 1.2x - 0.2) \\ \frac{\partial}{\partial x}\left(x^2 + y^2 - 1\right) & \frac{\partial}{\partial y}\left(x^2 + y^2 - 1\right) \end{pmatrix} = \begin{pmatrix} \cos(x + y) - 1.2 & \cos(x + y) \\ 2x & 2y \end{pmatrix}$$

Створимо клас sympy_matrix. Це буде матриця елементами якої будуть функції створені за допомогою бібліотеки sympy.

In [22]:

```
class sympy_matrix:

    def __init__(self, dimentions, list_of_func):
        self.matrix = np.reshape(list_of_func,dimentions)
        self.dimentions = dimentions

    def calculate(self, vector, inversed = False):
        array = np.zeros(self.dimentions)

        for i in range(self.dimentions[0]):
            for j in range(self.dimentions[1]):

                array[i][j] = float(self.matrix[i][j].subs([(x, vector[0]), (y, vector[1])]))

        if inversed:
            return np.linalg.inv(array)

        return array
```

In [36]:

```
def newton_method(jakobi_matrix, F, x_0):

    x_0 = np.array([[x_0[0]], [x_0[1]]])
    vector = (x_0[0][0], x_0[1][0])
    x_1 = x_0 - np.matmul(jakobi_matrix.calculate(vector, inversed = True), F.calculate(vector))
    vector_1 = (x_1[0][0], x_1[1][0])
    data = np.vstack((np.array([[vector[0], vector[1]]]), np.array([[vector_1[0], vector_1[1]]])))
    deltas = np.vstack((np.array([[0]]), np.array([[delta(vector, vector_1)]])))

    while delta((x_0[0][0], x_0[1][0]),(x_1[0][0], x_1[1][0])) > 0.00001:

        x_0 = x_1
        vector = (x_0[0][0], x_0[1][0])
        x_1 = x_0 - np.matmul(jakobi_matrix.calculate(vector, inversed = True), F.calculate(vector))
        vector_1 = (x_1[0][0], x_1[1][0])
        data = np.vstack((data, np.array([[vector_1[0], vector_1[1]]])))
        deltas = np.vstack((deltas, np.array([[delta(vector,vector_1)]])))

    iter_info = np.hstack((data, deltas))

    return x_1, iter_info
```

In [90]:

```
from sympy import *

x, y = symbols('x y')

f_1 = sin(x+y) - 1.2*x - 0.2
f_2 = x*x + y*y - 1

partial_f_1_x = cos(x+y) -1.2
partial_f_1_y = cos(x+y)
partial_f_2_x = 2*x
partial_f_2_y = 2*y


jacobi_matrix = sympy_matrix((2,2), [partial_f_1_x,partial_f_1_y,partial_f_2_x,partial_f_2_y])
F = sympy_matrix((2,1), [f_1, f_2])
x_0 = (-1, -0.25)

RResult, new_data = newton_method(jacobi_matrix, F, x_0)
```

Результати роботи програми оформити у звіті у вигляді таблиці.

Результати роботи програми записані в масив new_data. Перетворимо цей масив в об'єкт типу pandas.DataFrame, щоб його можна було вивести у вигляді таблиці. Для цього вже була створена фунція to_df()

In [72]:

```
to_df(new_data)
```

Out[72]:

	x	y	delta
1	-1.000000	-0.250000	0.000000
2	-0.957860	-0.293559	0.043559
3	-0.956832	-0.290659	0.002900
4	-0.956825	-0.290665	0.000006

Виконати перевірку, обчисливши $F(x\_^*)$

In [65]:

```
F.calculate((RResult[0][0], RResult[1][0]))
```

Out[65]:

```
array([[9.70445946e-13],
       [6.75174361e-11]])
```

Задати декілька інших початкових наближень (які не близькі до розв'язку) та з'ясувати як змінюється при цьому ітераційний процес, написати про це у висновку

Код аналогічний коду, який знаходиться у відповідному пункті попереднього завдання.

In [87]:

```
initial_approximations = (np.random.rand(5,2)-1)*100

new_data = np.zeros((5,5))

for i in range(5):

    new_data[i][0] = initial_approximations[i][0]
    new_data[i][1] = initial_approximations[i][1]

    result, info  = newton_method(jacobi_matrix, F, initial_approximations[i])

    new_data[i][2] = info.shape[0]
    new_data[i][3] = result[0][0]
    new_data[i][4] = result[1][0]

num = new_data.shape[0]
pd.DataFrame(new_data, columns = [ 'x_0','y_0','iter', 'x', 'y'], index = np.arange(1,num+1))
```

Out[87]:

	x_0	y_0	iter	x	y
1	-68.341096	-75.885760	46.0	0.656018	0.754745
2	-72.519021	-58.966651	21.0	-0.956825	-0.290665
3	-7.382951	-42.100851	67.0	-0.956825	-0.290665
4	-69.259106	-32.618426	22.0	0.656018	0.754745
5	-14.052467	-20.919615	32.0	-0.956825	-0.290665

Як бачимо, при певних значеннях початкового наближення кількість ітерацій сильно збільшується. При деяких значеннях початкового наближення наш метод знаходить другий корінь рівняння, але це не протирічить минулому факту.

Знайти розв'язок системи за допомогою fsolve бібліотеки scipy.optimize

In [93]:

```
def Func(x):
    return (np.sin(x[0]+x[1]) -1.2*x[0] - 0.2, x[0]**2 + x[1]**2 - 1)

x_0 = (-1, -0.25)
print("розв'язок отриманий за допомогою fsolve:", tuple(fsolve(Func, x_0)))
print("розв'язок отриманий за допомогою МН    :", (RResult[0][0], RResult[1][0]))
```

```
розв'язок отриманий за допомогою fsolve: (-0.9568250258901118, -0.29066453142845994)
розв'язок отриманий за допомогою МН    : (-0.9568250259097143, -0.29066453147983495)
```

*МН - метод Ньютона

Висновок.

Під час виконання цієї лабораторної роботи були розглянуті два методи пошуку коренів системи нелінійних рівнянь, а саме метод простих ітерацій і метод Ньютона. Було показано, що в випадку, коли початкове наближення не є близьким до розв'язку у методі простих ітерацій кількість ітерацій збальшується, але не сильно, а от у методі Ньютона значно збільшується.

In [ ]: