Technical University of Moldova

Homework nr.2

on Numerical Analysis

executed in Python programming language

by the student from FAF – 213 academic group

Bajenov Sevastian

Chisinau - 2022

Problem 2.1:

Numerical Analysis

Home work 2

(1)

Problem 2.1

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \quad , x \in [-3;3]$$

we know the Taylor series for e^t centered at 0:

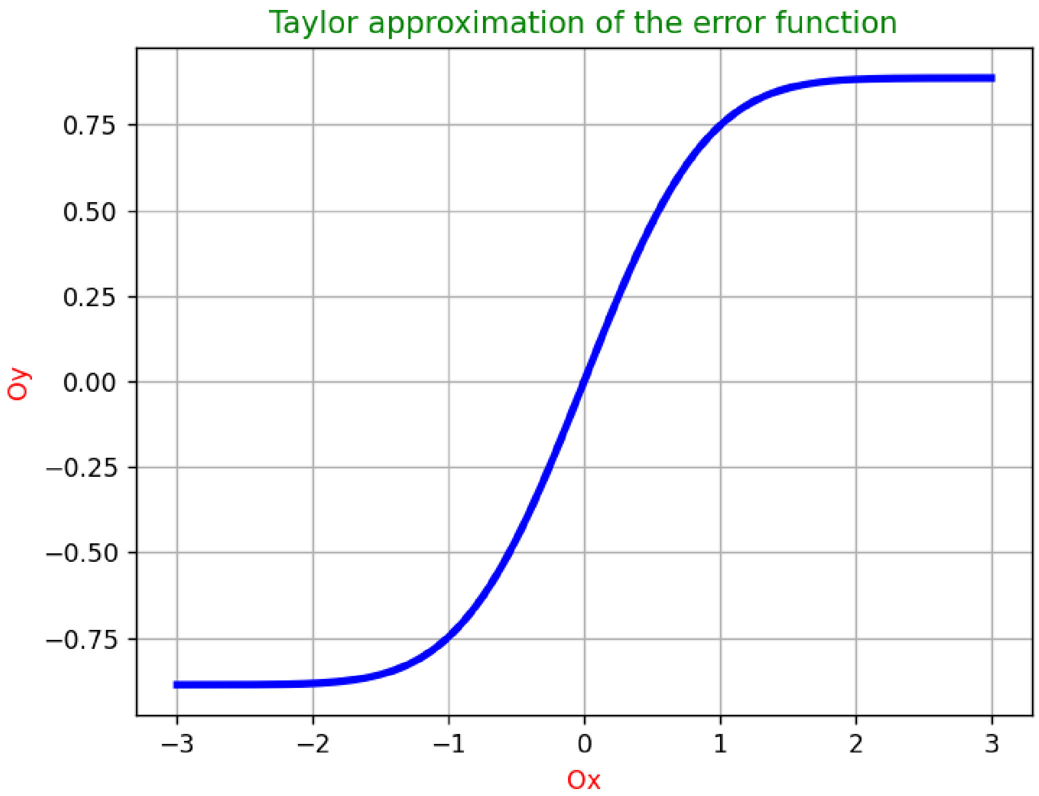
$$e^t = \sum_{n=0}^{\infty} \frac{t^n}{n!} \Rightarrow e^{-t^2} = \sum_{n=0}^{\infty} \frac{(-1)^n t^{2n}}{n!}$$

integrating the last expression term-by-term we obtain:

$$\frac{\sqrt{\pi}}{2} erf(x) = \sum_{n=0}^{\infty} \frac{(-1)^n t^{2n+1}}{(2n+1)\, n!} \,, \quad \frac{2}{\sqrt{\pi}} \cdot \frac{3^{2n+1}}{(2n+1)\, n!} \leq 10^{-5}$$

finally:
$$erf(x) = \frac{2}{\sqrt{\pi}}\left(t - \frac{t^3}{3} + \frac{t^5}{10} - \frac{t^7}{42} + \dots\right) = \frac{2}{\sqrt{\pi}}\left(x - \frac{x^3}{3} + \frac{x^5}{10} - \frac{x^7}{42} + \dots\right)$$

if we try to compute the 30-th term of the series the result will be $\approx 7{,}86 \cdot 10^{-6}$ which corresponds to the error of 10^{-5}.



Solution:

Error function.py

```python
import matplotlib.pyplot as plot
import math

# x_args will contain the domain of our function
# and the y_args list will store its values
x_args = list()
i = -3
while i < 3:
    x_args.append(i)
    i += 0.01

y_args = list()
for j in range(0, len(x_args)):
    sum = 0
    for i in range(0, 30):
        if i % 2 == 0:
            sum += pow(x_args[j], 2 * i + 1) / ((2 * i + 1) * math.factorial(i))
        else:
            sum -= pow(x_args[j], 2 * i + 1) / ((2 * i + 1) * math.factorial(i))
    y_args.append(sum)


graph = plot.figure()
plot.plot(x_args, y_args, color='blue', linewidth=3)
plot.xlabel("Ox", color='red')
plot.ylabel("Oy", color='red')
plot.title("Taylor approximation of the error function", color='green')
plot.grid()
plot.show()
```

Problem 2.2:

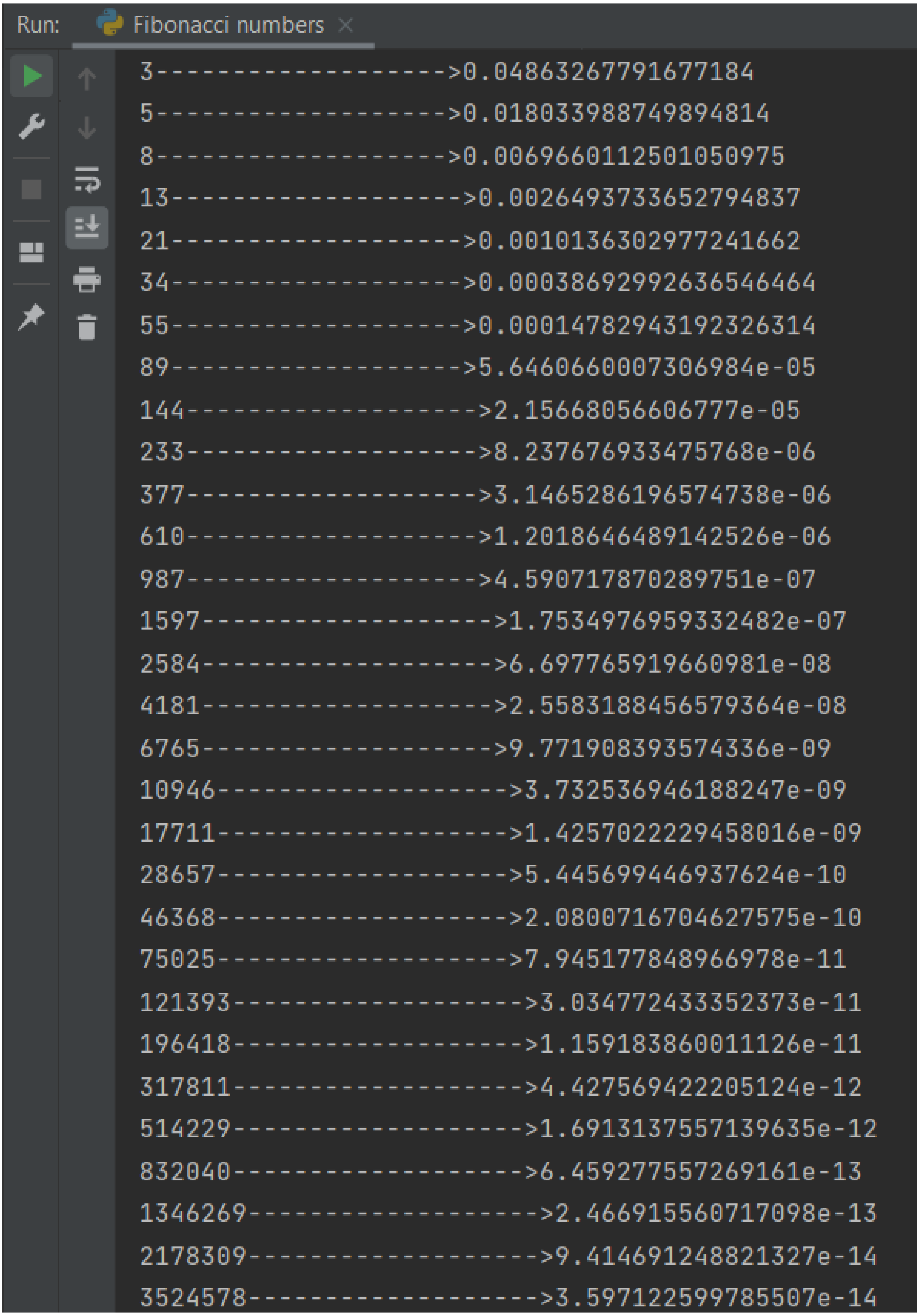
Solution and output:

Fibonacci numbers.py

```python
import math

# Python uses IEEE double precision
# therefore we can use it for solving this problem

# computing the Fibonacci sequence
# and the errors
fibonacci = list()
fibonacci.append(1)
fibonacci.append(1)

golden_ratio = (1 + math.sqrt(5)) / 2
errors = list()
for i in range(2, 40):
    fibonacci.append(fibonacci[i-2] + fibonacci[i - 1])

for i in range(0, 39):
    errors.append(abs(golden_ratio - fibonacci[i + 1] / fibonacci[i]))

print("Fibonacci numbers    Errors")
print("--------------------------")
for i in range(0, len(fibonacci) - 1):
    print(f"{fibonacci[i]}------------------>{errors[i]}")
print(fibonacci[len(fibonacci) - 1])

# the sequence of Fibonacci numbers has linear convergence

```

Run: Fibonacci numbers

```
C:\Users\User\PycharmProjects\Laboratories\
Fibonacci numbers    Errors
--------------------------
1------------------>0.6180339887498949
1------------------>0.3819660112501051
2------------------>0.1180339887498949
3------------------>0.04863267791677184
5------------------>0.018033988749894814
```



Run:
Fibonacci numbers
3------------------->0.04863267791677184
5------------------->0.018033988749894814
8------------------->0.0069660112501050975
13------------------->0.0026493733652794837
21------------------->0.0010136302977241662
34------------------->0.00038692992636546464
55------------------->0.00014782943192326314
89------------------->5.6460660007306984e-05
144------------------->2.1566805660677e-05
233------------------->8.237676933475768e-06
377------------------->3.1465286196574738e-06
610------------------->1.2018646489142526e-06
987------------------->4.590717870289751e-07
1597------------------->1.7534976959332482e-07
2584------------------->6.697765919660981e-08
4181------------------->2.5583188456579364e-08
6765------------------->9.771908393574336e-09
10946------------------->3.732536946188247e-09
17711------------------->1.4257022229458016e-09
28657------------------->5.445699446937624e-10
46368------------------->2.0800716704627575e-10
75025------------------->7.945177848966978e-11
121393------------------->3.034772433352373e-11
196418------------------->1.159183860011126e-11
317811------------------->4.427569422205124e-12
514229------------------->1.6913137557139635e-12
832040------------------->6.459277557269161e-13
1346269------------------->2.466915560717098e-13
2178309------------------->9.414691248821327e-14
3524578------------------->3.597122599785507e-14

```
5702887------------------->1.3766765505351941e-14
9227465------------------->5.329070518200751e-15
14930352------------------->1.9984014443252818e-15
24157817------------------->8.881784197001252e-16
39088169------------------->2.220446049250313e-16
63245986------------------->2.220446049250313e-16
102334155

Process finished with exit code 0
```

Problem 2.3:

Solution and output:

Newton's thermistors.py

```python
import math

# in this problem we will apply Newton;s method of solving equations
# for that purpose we wil need to introduce the function and its derivative

# r and t represent resistance and the temperature respectively
a = 8.775468 * pow(10, -8)
b = 2.341077 * pow(10, -4)
c = 1.129241 * pow(10, -3)


def function(r, t):
    return a * pow(math.log(r, math.e), 3) + b * math.log(r, math.e) + c - 1 / t


def derivative(r):
    return (3 * a * pow(math.log(r, math.e), 2)) / r + b / r


# the precision of the root has to be 10^-5
# we will calculate R for the temperatures
# 18.99 + 273.15 and 19.01 + 273.15 Kelvin
```

```python
def Newton_method(r0, t, error):
    resistance = list()
    resistance.append(r0)
    resistance.append(r0 - function(r0, t) / derivative(r0))
    index = 1
    while abs(resistance[index] - resistance[index - 1]) > error:
        resistance.append(resistance[index] - function(resistance[index], t) / derivative(resistance[index]))
        index += 1

    print(resistance[index])


print("The obtained results for 18.99 + 273.15 Kelvin and 19.01 + 273.15 Kelvin are:")
Newton_method(15000, 18.99 + 273.15, pow(10, -5))
Newton_method(15000, 19.01 + 273.15, pow(10, -5))

# the obtained range is 13066.542623383364 <= R <= 13078.426633664925
```

Run: Newton's thermistors

```
C:\Users\User\PycharmProjects\Laboratories\venv\Scripts\python.exe "C:/Users/U
The obtained results for 18.99 + 273.15 Kelvin and 19.01 + 273.15 Kelvin are:
13078.426633664925
13066.542623383364

Process finished with exit code 0
```

Problem 2.4:

Solution and output:

Different convergences (Problem 4).py

```python
import math
import matplotlib.pyplot as plot


# in this problem we will use Newton's method and fixed - point iteration method
# also we ill try to optimize them to increase the order of convergence

# first of all we introduce the function and its derivative


def function(x):
    return pow(math.e, x - math.pi) + math.cos(x) - x + math.pi
```

```python
def derivative(x):
    return pow(math.e, x - math.pi) - math.sin(x) - 1


def second_derivative(x):
    return pow(math.e, x - math.pi) - math.cos(x)


# creating the domain and the range of the function
x_args = list()
y_args = list()
i = 0
while i < 5:
    x_args.append(i)
    y_args.append(function(i))
    i += 0.01

# plotting the graph of the function
graph = plot.figure()
plot.plot(x_args, y_args, color='red', linewidth=3)
plot.xlabel("x-axis", color='blue')
plot.ylabel("y-axis", color='blue')
plot.title("Function f", color='green')
plot.grid()
plot.show()
```

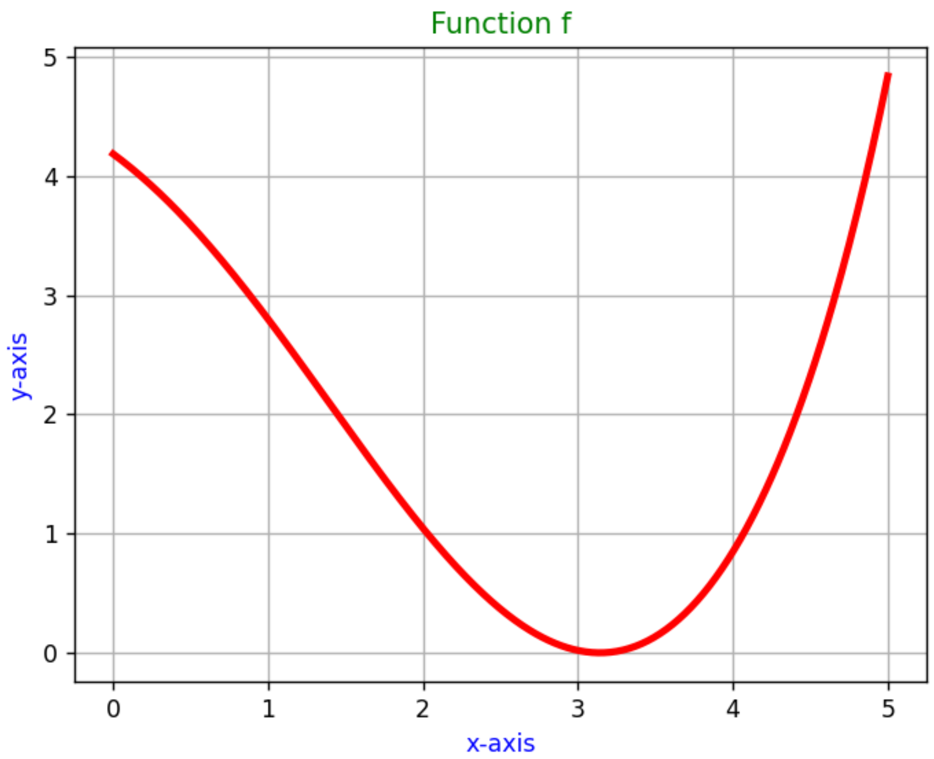
```python
# now we start applying Newton's method
# our initial guess will be 2 / b or 2 / 5
# with the accuracy 10^-5
def Newton_method(x0, error):
    x_list = list()
    x_list.append(x0)
    x_list.append(x0 - function(x0) / derivative(x0))
```

```python
    print("Newton's method:")
    print("n   xn                    xn - xn-1             order of convergence")
    print("----------------------------------------------------------------------")
    print(f"{0}   {x0}")
    print(f"{1}   {x_list[1]}   {x_list[1] - x0}")

    index = 1
    while abs(x_list[index] - x_list[index - 1]) > error:
        x_list.append(x_list[index] - function(x_list[index]) / derivative(x_list[index]))
        print(
            f"{index + 1}   {x_list[index + 1]}   {abs(x_list[index + 1] - x_list[index])}   {math.log(x_list[index], x_list[index + 1])}")
        index += 1

    for j in range(0, 3):
        print()


Newton_method(0.4, pow(10, -5))


# the convergence in this case is linear, because the solution
# is situated at the point where the tangent line is Ox axis
# and the derivative is equal to zero
```

```python
def fixedPoint_method(x):
    return pow(math.e, x - math.pi) + math.cos(x) + math.pi


# now we will try to use fixed - point iteration method
x0 = 3
x_args = list()
x_args.append(x0)

errors = list()
for i in range(1, 21):
    x_args.append(fixedPoint_method(x_args[i - 1]))
    errors.append(abs(x_args[i] - x_args[i - 1]))

convergence = list()
for i in range(1, len(errors)):
    convergence.append(math.log(x_args[i - 1], x_args[i]))

print("Fixed point iteration:")
print("n   xn                    xn - xn-1             order of convergence")
print("----------------------------------------------------------------------")
print(f"{0}   {x0}")
for i in range(1, 21):
    if i < 2:
        print(f"{i}   {x_args[i]}   {errors[i - 1]}")
    else:
        print(f"{i}   {x_args[i]}   {errors[i - 1]}   {convergence[i - 2]}")
```

```python
for j in range(0, 3):
    print()


# according to the results we can say that the order of convergence
# tends to 0.93 that is why this is the linear convergence, which is
# slower than Newton's method and highly depends on the initial guess
# which I have chosen according to the drawn graph
```

```python
# if we take the derivative of the initial function and apply Newton's method
# to it, then we will have a quadratic convergence
def Newton_method_upgrade(x0, error):
    x_list = list()
    x_list.append(x0)
    x_list.append(x0 - derivative(x0) / second_derivative(x0))

    print("Newton's method upgraded:")
    print("n   xn                     xn - xn-1              order of convergence")
    print("----------------------------------------------------------------------")
    print(f"{0}   {x0}")
    print(f"{1}   {x_list[1]}   {x_list[1] - x0}")

    index = 1
    while abs(x_list[index] - x_list[index - 1]) > error:
        x_list.append(x_list[index] - derivative(x_list[index]) / second_derivative(x_list[index]))
        print(f"{index + 1}   {x_list[index + 1]}   {abs(x_list[index + 1] - x_list[index])}   {math.log(x_list[index], x_list[index + 1])}")
        index += 1

    for j in range(0, 3):
        print()


Newton_method_upgrade(4, pow(10, -5))
```



Run: Different convergences (Problem 4)

```
C:\Users\User\PycharmProjects\Laboratories\venv\Scripts\python.exe "C:/
Newton's method:
n   xn                     xn - xn-1              order of convergence
------------------------------------------------------------------------
0   0.4
1   3.2130262450788876   2.8130262450788877
2   3.1775184972939425   0.03550774778494503   1.00961222892545
3   3.1596088849309236   0.017909612363018912   1.004913124998217
4   3.150614233636119   0.0089946512948047   1.002484164431146
5   3.146106827227523   0.004507406408595926 5   1.001249093367068 4
6   3.143850588527611   0.002256238699911872   1.000626313599669
7   3.142721833366968   0.0011287551606429247   1.0003136003137447
8   3.1421572965904585   0.0005645367765096942   1.0001569112596302
9   3.141874988372613   0.00028230821784536175   1.0000784834336174
10   3.1417338243026554   0.00014116406995778874   1.0000392486712262
11   3.141663239774427   7.058452521269132e-05   1.0000196260745178
12   3.141627946890095   3.529288734771541e-05   1.0000098134726303
13   3.14161030029644 6   1.764659364900112e-05   1.0000049068434362
14   3.141601476951199   8.823345246877778e-06   1.0000024534515377
```

Run: Different convergences (Problem 4)

```
Fixed point iteration:
n   xn                     xn - xn-1              order of convergence
-------------------------------------------------------------------
0   3
1   3.0195749078719594   0.01957490787195937
2   3.0341602484235755   0.014585340551616088   0.9941148499699608
3   3.045495195927971 5   0.011334947504396009   0.9956586323590778
4   3.0545819456353374   0.009086749707365893   0.9966517563519985
5   3.0620429775421023   0.007461031906764948   0.997331978600956
6   3.0682872022614727   0.0062442247193703615   0.9978200063992201
7   3.07359522079269   0.005308018531217318   0.9981829248014027
8   3.078166460420912   0.004571239628222035   0.9984606404075254
9   3.0821468080120527   0.003980347591140632   0.9986781895752423
10   3.085645598798042   0.0034987907859891365   0.9988519679859664
11   3.0887464808612686   0.003100882063226784   0.9989930996352191
12   3.0915145980495264   0.002768117188257868   0.9991093597166921
13   3.094001476060163   0.0024868780106364063   0.9992063224325759
14   3.0962484292001298   0.0022469531399669 23   0.9992880726198725
15   3.0982889876122144   0.0020405584120846143   0.9993576615364379
16   3.1001506599555193   0.001861672343304921   0.9994174071476375
17   3.101856235439069   0.0017055754835495485   0.9994690969267682
18   3.1034247603641933   0.0015685249251244926   0.9995141278521207
19   3.104872280682424   0.0014475203182304774   0.9995536049560622
20   3.1062124137144975   0.0013401330320736982   0.9995884119240812
```

```
Newton's method upgraded:
n   xn                         xn - xn-1                order of convergence
----------------------------------------------------------------------
0   4
1   3.2976528854289495   -0.7023471145710505
2   3.147276135094709   0.15037675033424058   1.0407083192235922
3   3.1416007062619786   0.0056754288327303115   1.001576709601429
4   3.1415926536060046   8.052655974033485e-06   1.0000022391629266



Process finished with exit code 0
```

Problem 2.5:

Solution and output:

Fixed point iteration VS bisection method.py

```python
import math

# first of all we simulate fixed point iteration
x0 = 0.1
x_args = list()
x_args.append(x0)

errors = list()
for i in range(1, 11):
    x_args.append(math.cos(x_args[i - 1]) - 1 + x_args[i - 1])
    errors.append(abs(x_args[i] - x_args[i - 1]))

convergence = list()
for i in range(1, len(errors)):
    convergence.append(errors[i] / errors[i - 1])


print("Fixed point iteration:")
print("n   xn                        xn - xn-1              rate of convergence")
print("---------------------------------------------------------------------")
print(f"{0}   {x0}")
for i in range(1, 11):
    if i < 2:
        print(f"{i}   {x_args[i]}   {errors[i - 1]}")
    else:
        print(f"{i}   {x_args[i]}   {errors[i - 1]}   {convergence[i - 2]}")
```

Fixed point iteration VS bisection method.py

```python

# in this case the convergence (lambda tends to 0.95) is linear
# and it is slower than the bisection method
# now let's try to use Aitken's extrapolation formula
x_args = list()
x_args.append(x0)

index = 0
for i in range(0, 3):
    x = x_args[index]
    x1 = math.cos(x) - 1 + x
    x2 = math.cos(x1) - 1 + x1

    x_args.append(x1)
    x_args.append(x2)

    coefficient = (x2 - x1) / (x1 - x)
    x3 = x2 + (coefficient * (x2 - x1)) / (1 + coefficient)

    x_args.append(x3)
    index += 3

x_args.append(math.cos(x3) - 1 + x3)
```

```python
errors = list()
for i in range(1, len(x_args)):
    errors.append(abs(x_args[i] - x_args[i - 1]))

convergence = list()
for i in range(1, len(errors)):
    convergence.append(errors[i] / errors[i - 1])

print()
print()
print("Aitken's extrapolation:")
print("n   xn                    xn - xn-1             rate of convergence")
print("--------------------------------------------------------------------")
print(f"{0}   {x0}")
for i in range(1, 11):
    if i < 2:
        print(f"{i}   {x_args[i]}   {errors[i - 1]}")
    else:
        print(f"{i}   {x_args[i]}   {errors[i - 1]}   {convergence[i - 2]}")

# in this case the speed of convergence increases
```

Run: Fixed point iteration VS bisection method

```
C:\Users\User\PycharmProjects\Laboratories\venv\Scripts\python.exe "C:/
Fixed point iteration:
n   xn                    xn - xn-1              rate of convergence
-------------------------------------------------------------------------
0   0.1
1   0.09500416527802583   0.004995834721974179
2   0.09049466291815525   0.004509502359870576   0.9026524316418095
3   0.08640281449600365   0.0040918484221516005   0.9073835859504988
4   0.08267241294550767   0.0037304015504959764   0.9116666028734354
5   0.07925699496704888   0.00341541797845879   0.9155630921300396
6   0.0761178031386234   0.0031391918284254805   0.9191237641262412
7   0.07322224162070703   0.0028955615179163674   0.9223907541096937
8   0.07054269080796352   0.002679550812743514   0.9253993728552196
9   0.06805558682675197   0.002487103981211547   0.9281794431302736
10   0.06574069904689253   0.0023148877798594425   0.9307563324038376
```

```
Aitken's extrapolation:
n   xn                    xn - xn-1              rate of convergence
-------------------------------------------------------------------------
0   0.1
1   0.09500416527802583   0.004995834721974179
2   0.09049466291815525   0.004509502359870576   0.9026524316418095
3   0.08835527413492952   0.0021393887832257263   0.4744179318462819
4   0.08445448556775295   0.00390078856717657   1.823319163754352
5   0.0808903247231764   0.0035641608445765582   0.9137026483740779
6   0.07918860622177407   0.0017017185014023273   0.4774527793805278
7   0.07605482667565436   0.00313377954611970 8   1.8415381530713035
8   0.07316405218282598   0.002890774492828374   0.9224562386361138
9   0.07177696575047551   0.0013870864323504706   0.4798321127405984
10   0.0692021050882182   0.0025748606622573167   1.8563087362148858

Process finished with exit code 0
```

Problem 2.6:

Problem 2.6

3

n	x_n	$x_n - x_{n-1}$	λ_n
0	2,0		
1	2,1248	0,124834	
2	2,2148	0,089944	0,720509
3	2,2805	0,065698	0,730432
4	2,3289	0,048386	0,736491
5	2,3647	0,035827	0,740441
6	2,3913	0,026624	0,743127
7	2,4111	0,019835	0,745005
8	2,4260	0,014803	0,746307
9	2,4370	0,011062	0,747281
10	2,4453	0,0082745	0,748011

$$\lambda_n = \frac{x_n - x_{n-1}}{x_{n-1} - x_{n-2}}$$

Obviously, λ_n tends to 0,75 and is higher than the bisection method with $\lambda \approx 0{,}5$;

The root α has multiplicity 4 because $\lambda_n = \frac{3}{4}$ and can be computed accurately by applying Newton's method for the equation: $f^{(3)}(x) = 0$;

This approach will speed up the convergence.

Problem 2.7:

Problem 2.7

a) $x = -\ln x$;

b) $x = e^{-x}$; for the equation $x + \ln x = 0$,

c) $x = \frac{x + e^{-x}}{2}$; which solution is $\alpha \approx 0{,}567143$

It means that the first formula cannot be used because of the domain of natural log ($x \in (0; +\infty)$) without negative numbers.

It is better to use the second formula than the third becau[se] it requires less operations with the iteration variable.

Personally, I would suggest to use the formula:

$$x = \cancel{*}\, \frac{x + \frac{1}{e^x}}{2} = \frac{xe^x + 1}{2e^x}; \text{ or } \frac{1}{e^x} = x;$$

Problem 2.8:

Problem 2.8

$$\lambda_n = \frac{x_n - x_{n-1}}{x_{n-1} - x_{n-2}};$$

n	x_n	$x_n - x_{n-1}$	λ_n
0	1.00		
1	0.36788	-6.3212E-01	
2	0.69220	3.2432E-01	+5.1306 E-1
3	0.50047	-1.9173E-01	+5.977 E-1
4	0.60624	1.0577 E-01	+5.5166 E-1
5	0.54540	-6.0848E-02	+5.7528 E-1
6	0.57961	3.4217E-02	+5.6233 E-1

a) λ is converging to 0,57 therefore the convergence is linear;

b) $g'(\alpha) \approx 0{,}57$, the rate of convergence is 0.57;
the rate of convergence of this method is a faster slower ~~little bit fas-ter~~ than the bisection method with ~~$\lambda = 0{,}5$~~; $\lambda = 0{,}5$ ~~$\approx \varphi$ (golden ratio)~~ ~~1.65~~

c) To accelerate the convergence of this method we can use Aitken's extrapolation formula;

$$\lambda = \frac{x_n - x_{n-1}}{x_{n-1} - x_{n-2}}; \quad \alpha - x_n = \frac{\lambda_n}{1 - \lambda_n}\left[x_n - x_{n-1}\right] \text{ for error estimation}$$

and $x_{n+1} = x_n + \dfrac{\lambda_n}{1+\lambda_n}\left[x_n - x_{n-1}\right]$ for computing the points.

Problem 2.9:

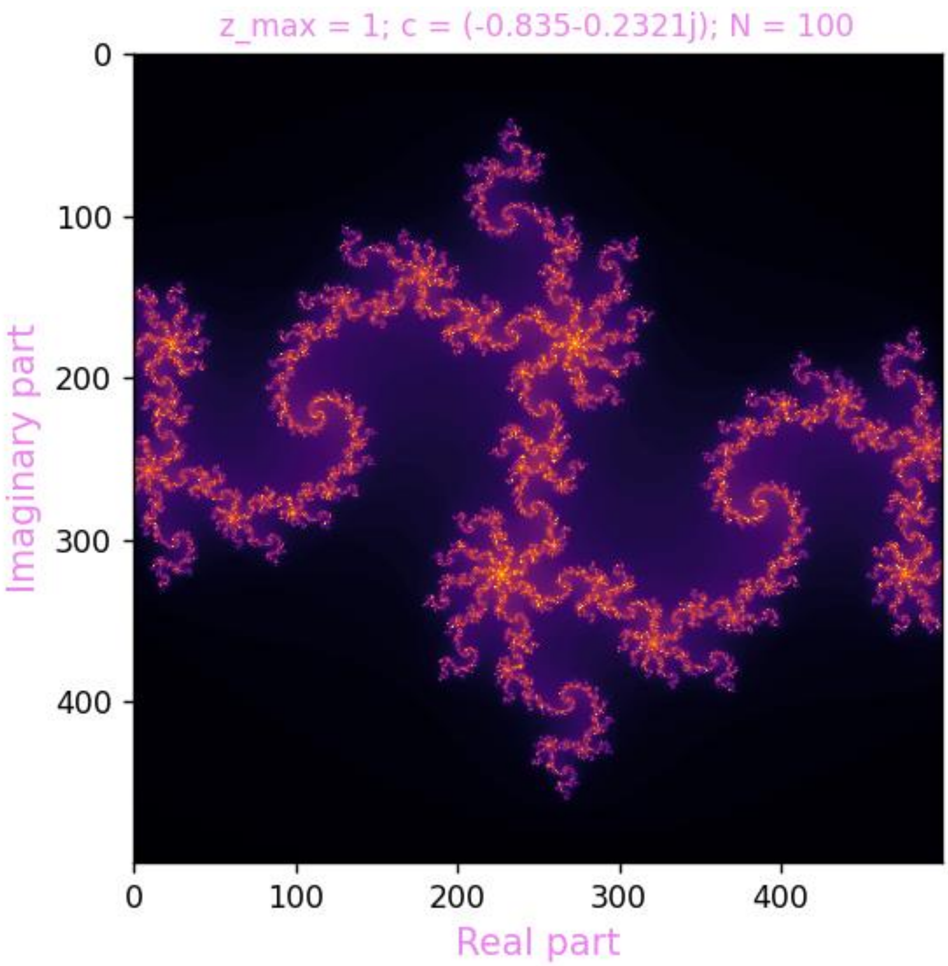
Solution and output:

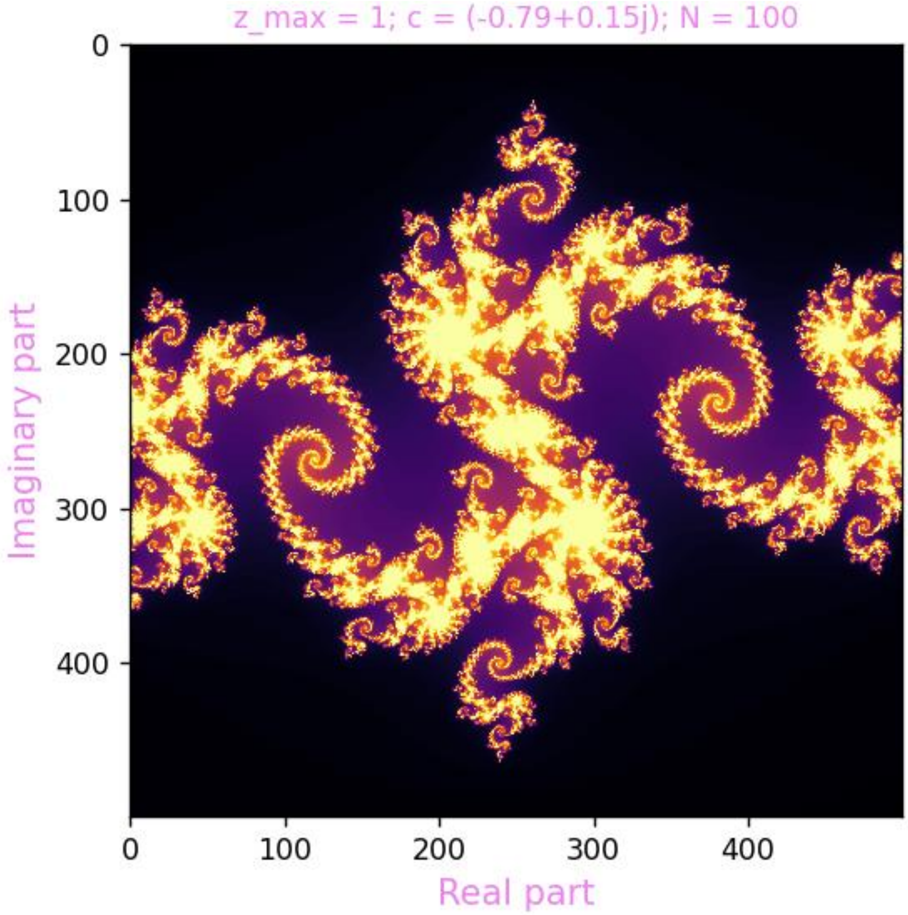
Fractals.py

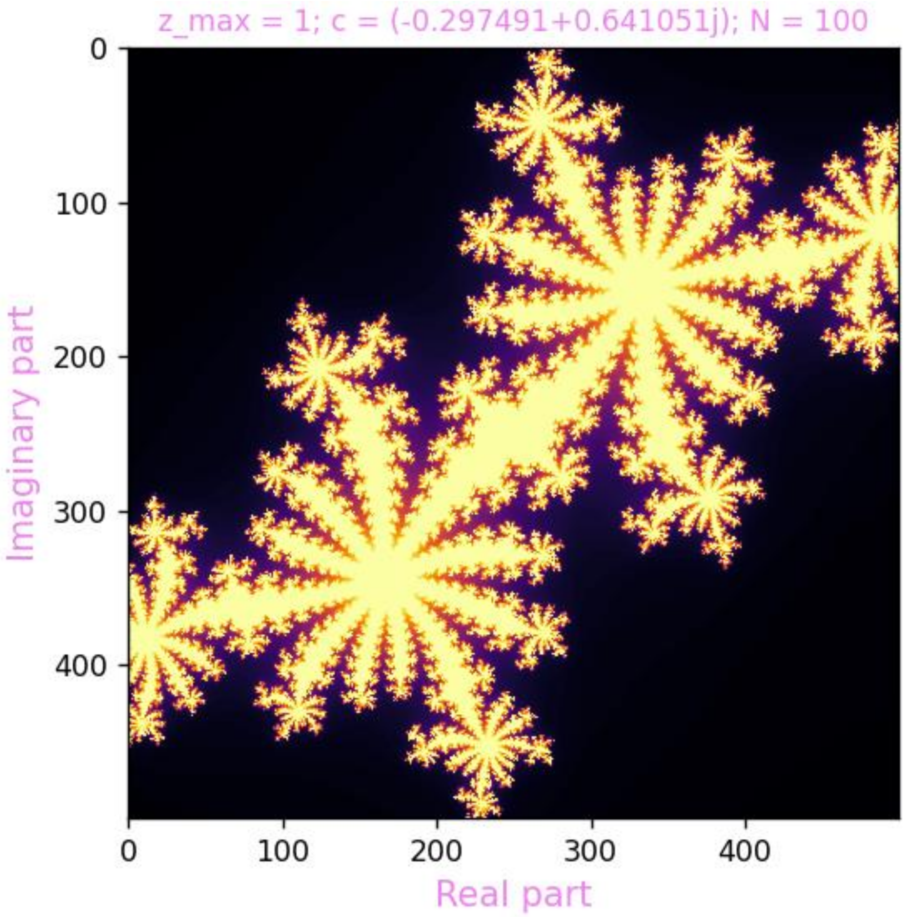
```python
import numpy as np
import matplotlib.pyplot as plot

# introducing basic parameters

# the bounds for the matrix of complex numbers
x_res = 500
y_res = 500

# the bounds for the real (x) and imaginary (y) parts of complex numbers
x_min, x_max = -1, 1
y_min, y_max = -1, 1

# z_max represents the stopping criteria for finding the escape velocity
z_max = 2

# maximum number of iterations (limit)
N = 100
```

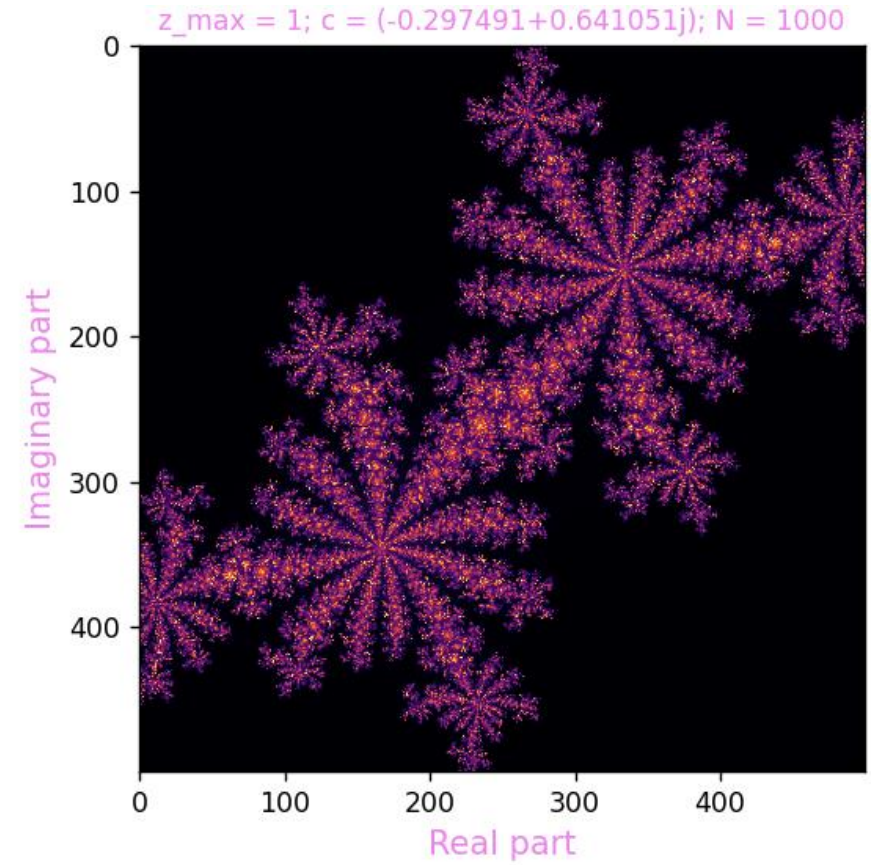
```python
# function for finding the escape velocity
def escVel(z0, c, N1):
    n = 0
    z = z0
    while (abs(z) <= z_max) and (n < N1):
        z = z ** 2 + c
        n += 1

    return n
```

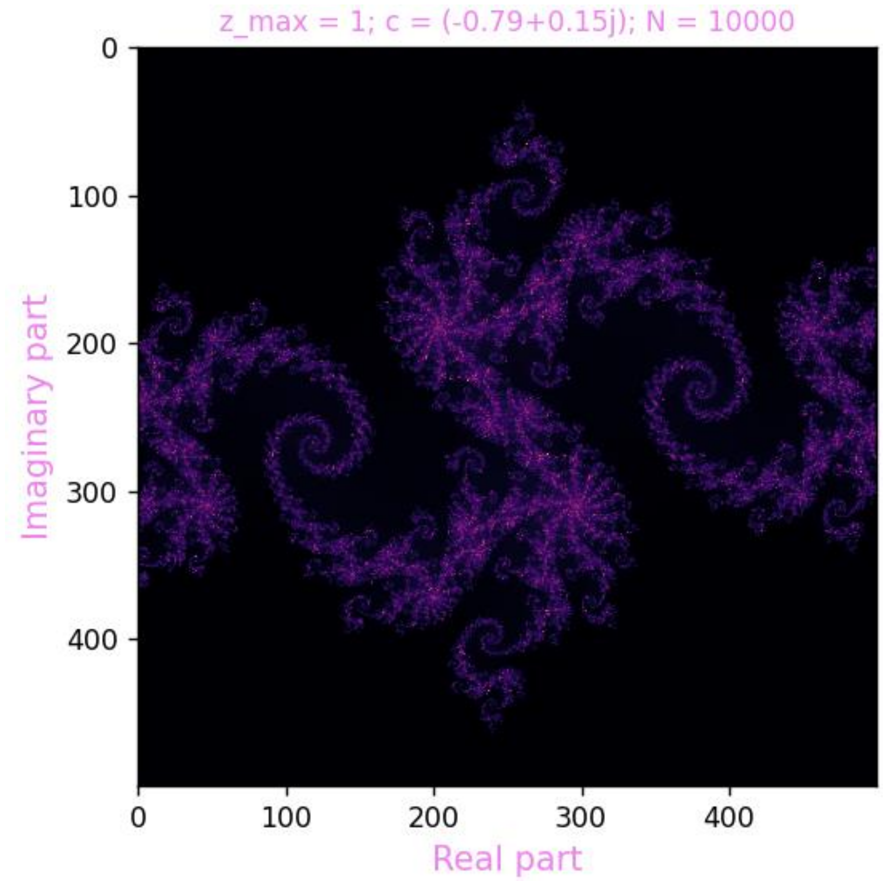
```python
# function for printing the Julia set
def JuliaSet(z_max1, c, N1):
    # initialising an empty matrix 500 * 500 for our complex numbers
    julia_set = np.zeros((x_res, y_res))

    for iy in range(0, y_res):
        for ix in range(0, x_res):
            # mapping each pixel (matrix element) to a position of a point in the complex plane
            z = complex(ix / x_res * (x_max - x_min) + x_min, iy / y_res * (y_max - y_min) + y_min)
            julia_set[iy][ix] = escVel(z, c, N1)

    # plotting the Julia matrix using matplotlib tool imshow
    plot.imshow(julia_set, cmap='inferno')
    plot.axis('on')
    plot.title(f'z_max = {z_max1}; c = {c}; N = {N1}', fontsize=10, color='violet')
    plot.xlabel('Real part', fontsize=12, color='violet')
    plot.ylabel('Imaginary part', fontsize=12, color='violet')
    plot.show()


# the following examples were taken from the homework file
JuliaSet(1, complex(-0.835, -0.2321), N)
JuliaSet(1, complex(-0.79, 0.15), N)
JuliaSet(1, complex(-0.297491, 0.641051), N)
```




z_max = 1; c = (-0.79+0.15j); N = 100
Imaginary part
Real part
0
100
200
300
400


z_max = 1; c = (-0.297491+0.641051j); N = 100
Imaginary part
Real part
0
100
200
300
400





Final remark: all the python files with the code will be uploaded together with this document;