NUMA01

Newton Fractals

Isak Nyström, Daniel Lizotte, Muhammad Nabeel Numan,
Samuel Silva, Adrian Evertsson, Josie Lindsey-Clark

Task 1
- Write the class fractal2D for a system of two equations with two variables.

Task 2
- Write the method __Newton__ which takes an initial guess as input.

Task 3
- Write the method __getzeroes__ to store a zero or a divergence given by __Newton__.

Task 4
- Write the method __plot__ to run __Newton__ for multiple initial guesses and visualize the results in a figure.

Task 5
- Write the method __simpNewton__ which will compute the jacobian only once.

Task 6
- Compute the derivatives numerically for the Jacobian and add these derivatives as an optional argument in __init__

Task 7
- Write the method __itPlot__ which show dependence between initial values and iterations needed to reach convergence.

Task 8
- Test the code with two further functions.

Task 1

Write a class fractal2D that is initialized with one function and possibly its derivative.

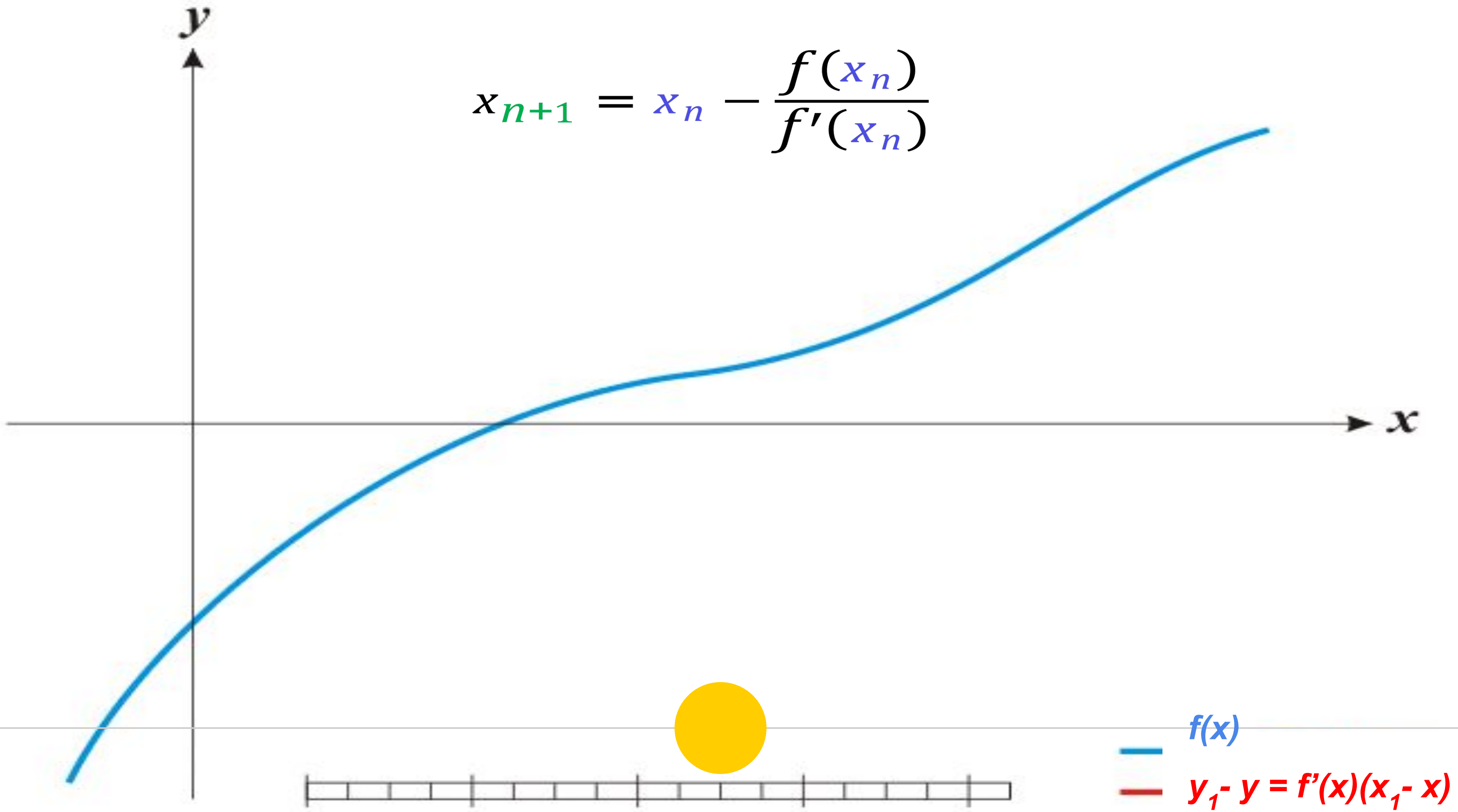
```python
class fractal2D(object):
    def __init__ (self, f, g):
        self.f = f
        self.g = g
        self.tol= 1.e-9
        self.listempty=True
        self.xz=[]
    def __call__(self, x):
        return f(x), g(x)

    def __repr__(self):
        return ("({},{})".format(self.f,self.g))
```

Task 2

Write a method __Newton__ which takes an initial guess as input

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$



Task 2

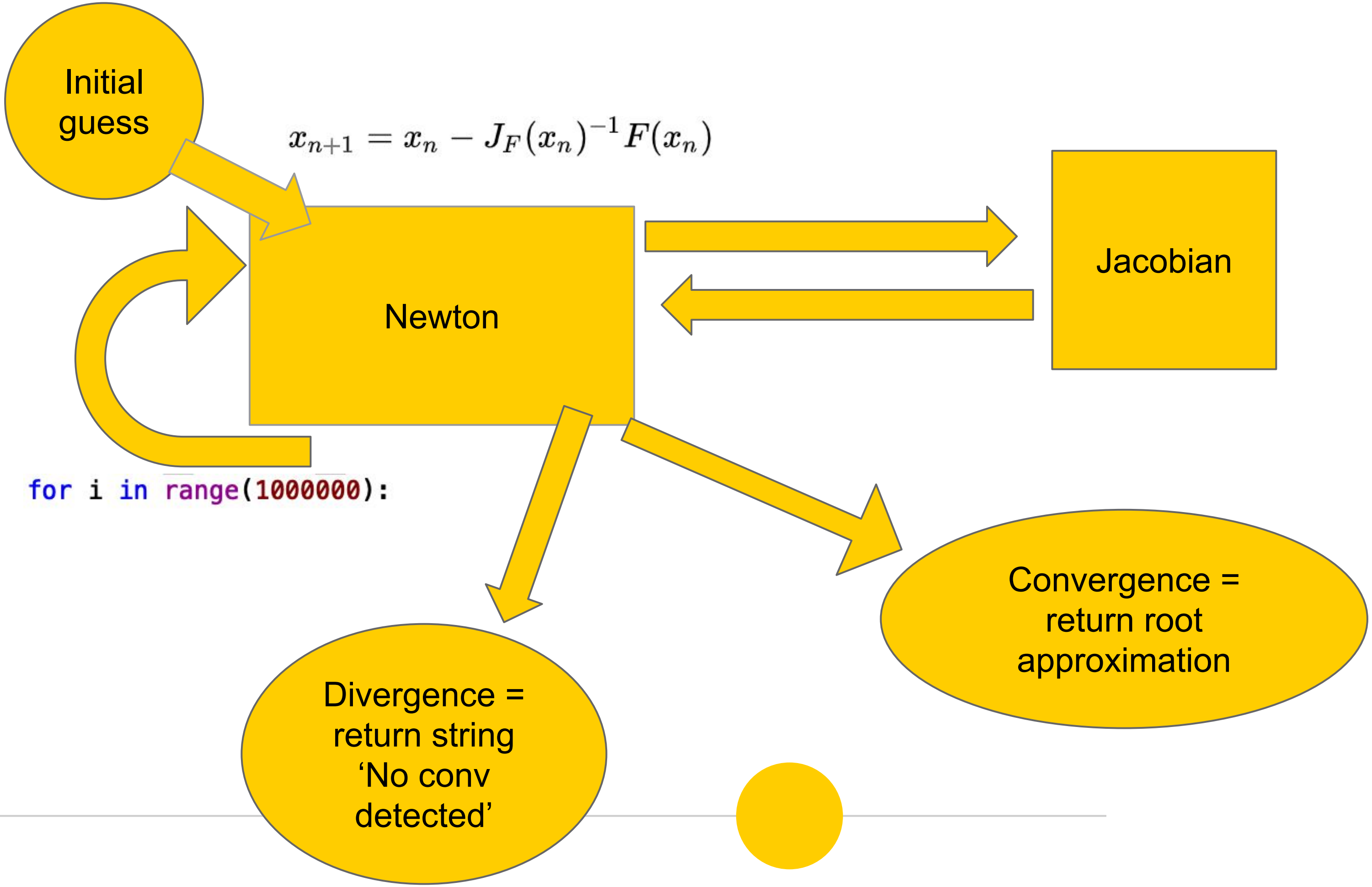
- For a matrix containing multiple variables, we use the equation:

$$x_{n+1} = x_n - J_F(x_n)^{-1} F(x_n)$$

$$J_F(x_n)(x_{n+1} - x_n) = -F(x_n)$$

- Where the Jacobian matrix is:

$$\mathbf{J_f}(x, y) = \begin{bmatrix} \dfrac{\partial f_1}{\partial x} & \dfrac{\partial f_1}{\partial y} \\ \\ \dfrac{\partial f_2}{\partial x} & \dfrac{\partial f_2}{\partial y} \end{bmatrix}$$

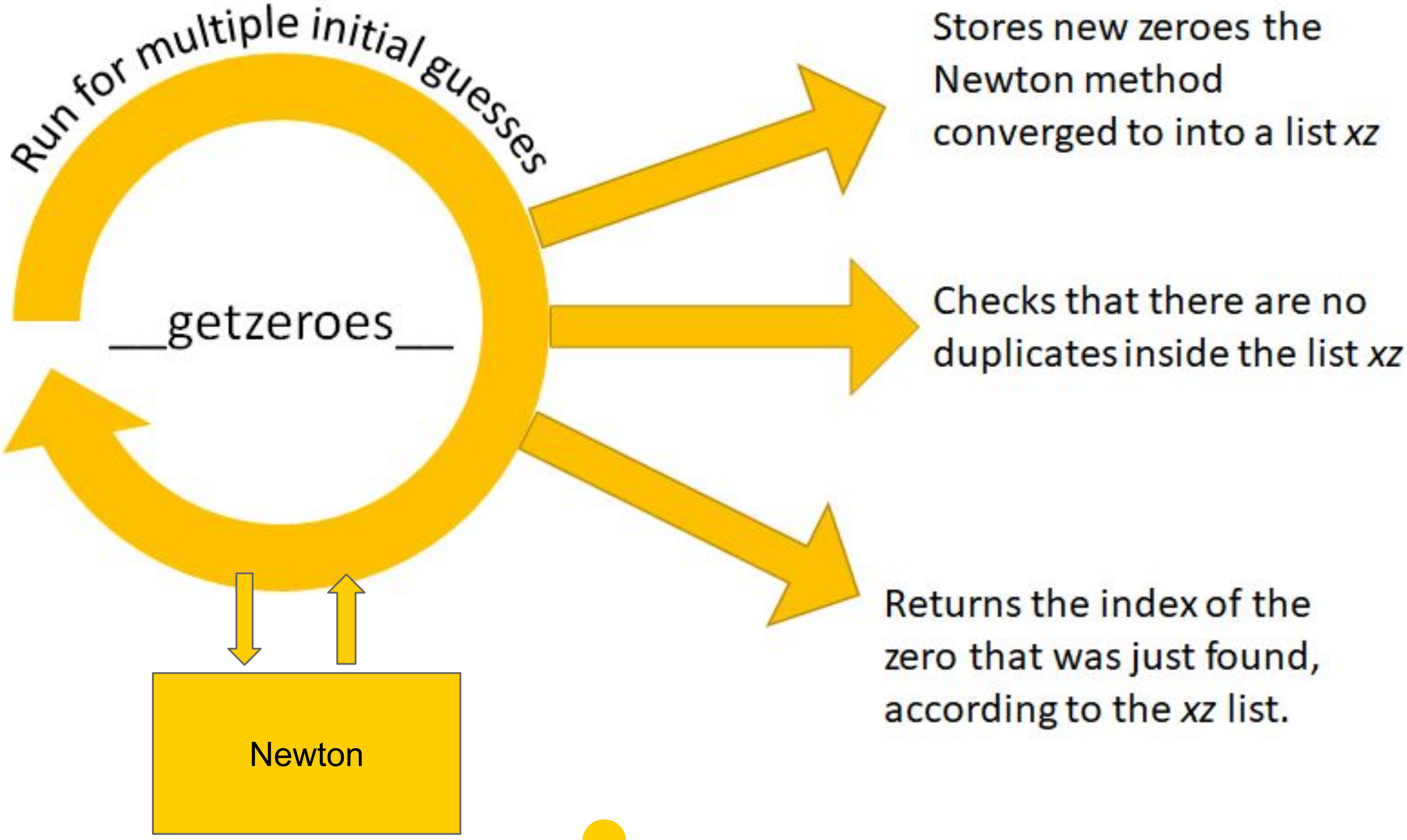

Initial guess
$x_{n+1} = x_n - J_F(x_n)^{-1} F(x_n)$
Newton
Jacobian
for i in range(1000000):
Convergence = return root approximation
Divergence = return string ‘No conv detected’

Task 3

Write the method __getzeroes__ to store a zero or a divergence given by __Newton__.

Task 3

- We made a method called __getzeroes__
- The function is called with an initial guess, runs them through the Newton method and checks if the root found in the newton method already exists in the list xz. If it is a new root, it will be stored in xz.
- The algorithm will return the index of the root that was found, or the value -1 if no convergence was detected

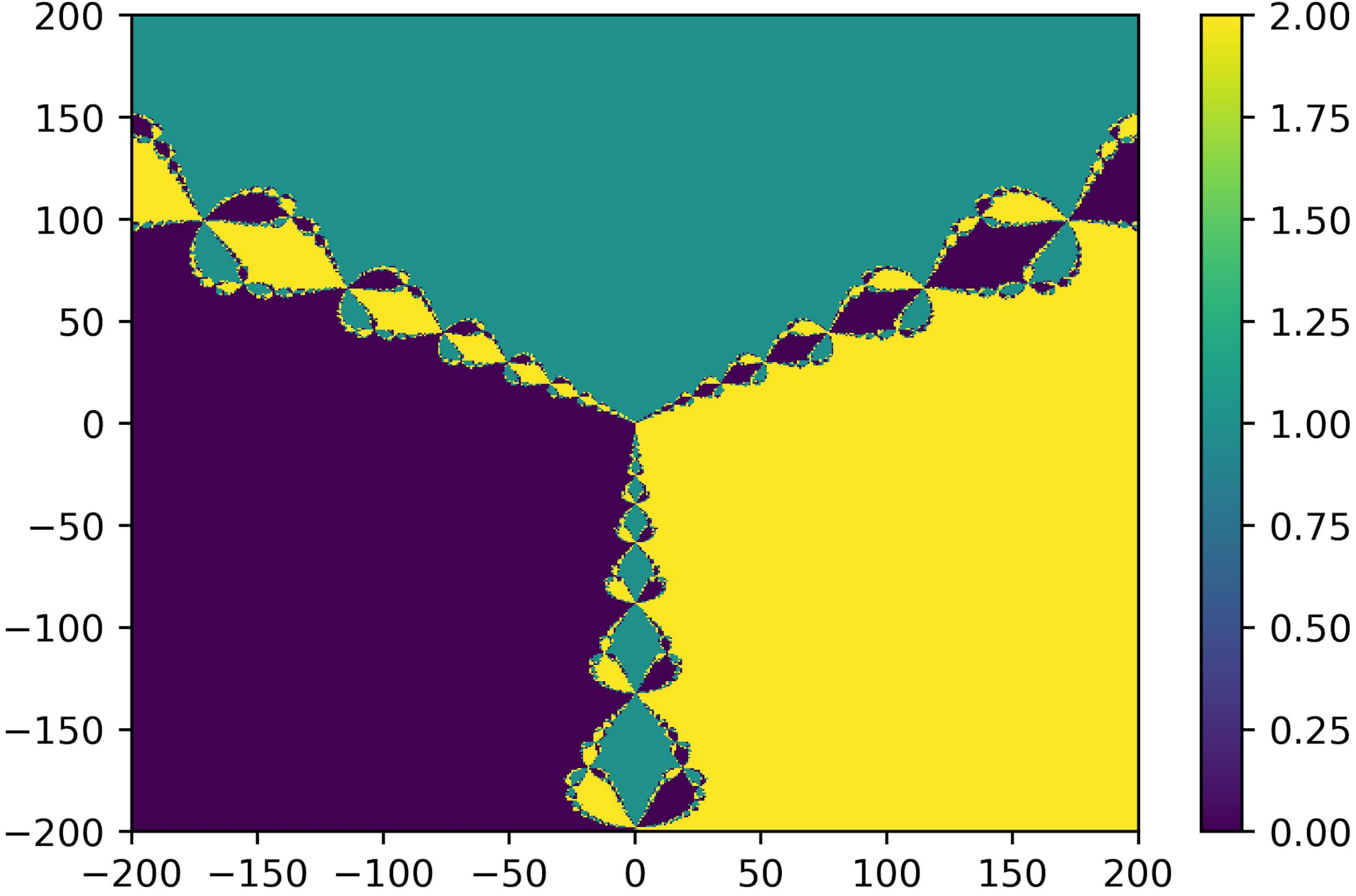

Run for multiple initial guesses
__getzeroes__
Newton
Stores new zeroes the Newton method converged to into a list xz
Checks that there are no duplicates inside the list xz
Returns the index of the zero that was just found, according to the xz list.

Task 4

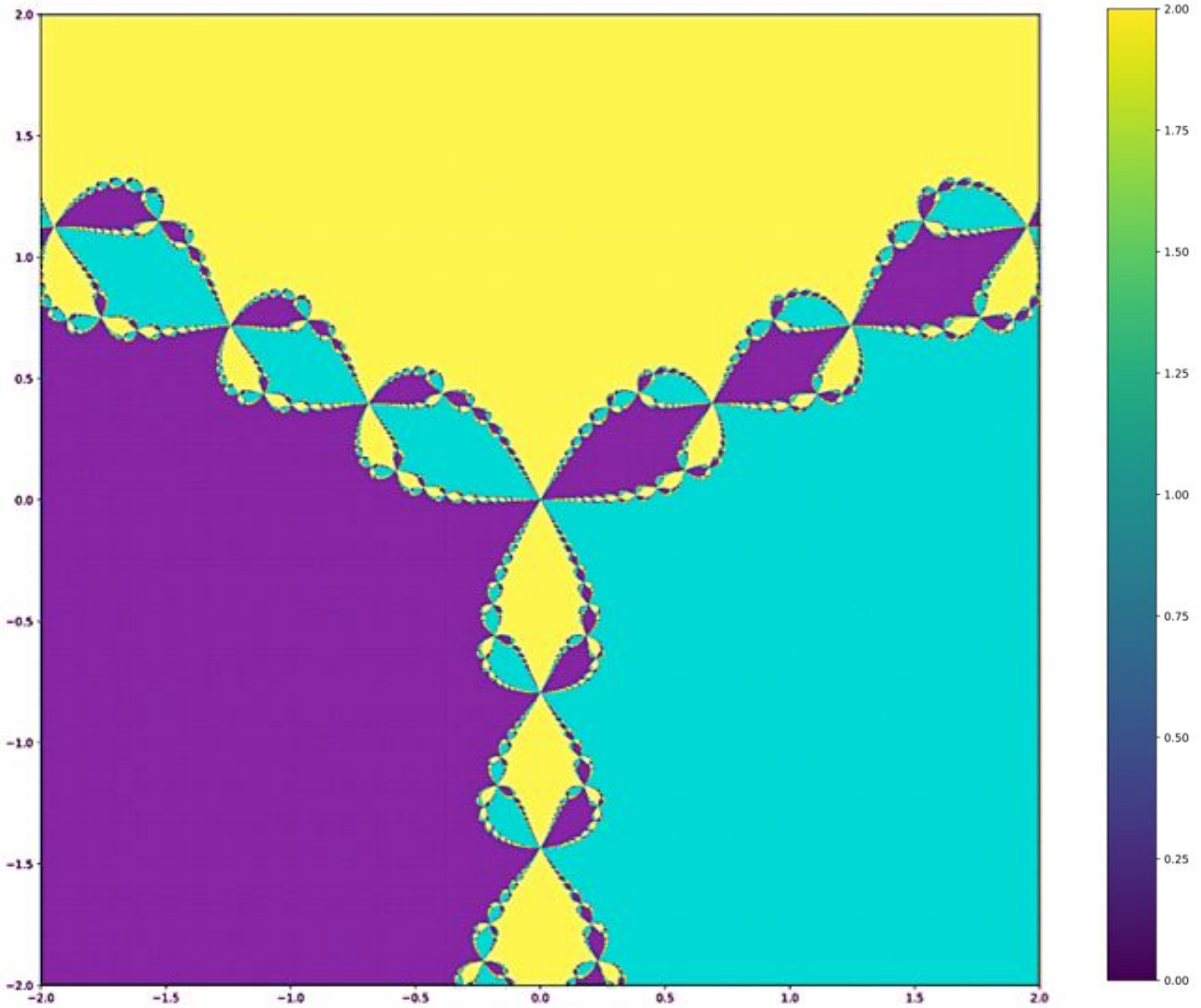
Write the method __plot__ to run __Newton__ for multiple initial guesses and visualize the results in a figure.

Task 4

- In this method we created a meshgrid with the values that we got from the input in our given NxN sized matrices
- We then use these matrices as our initial guesses when we call our getzeros function and then put the result in a new matrix called A
- Lastly we plot this matrix using pcolor to get our fractal



200
150
100
50
0
−50
−100
−150
−200
−200
−150
−100
−50
0
50
100
150
200
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00

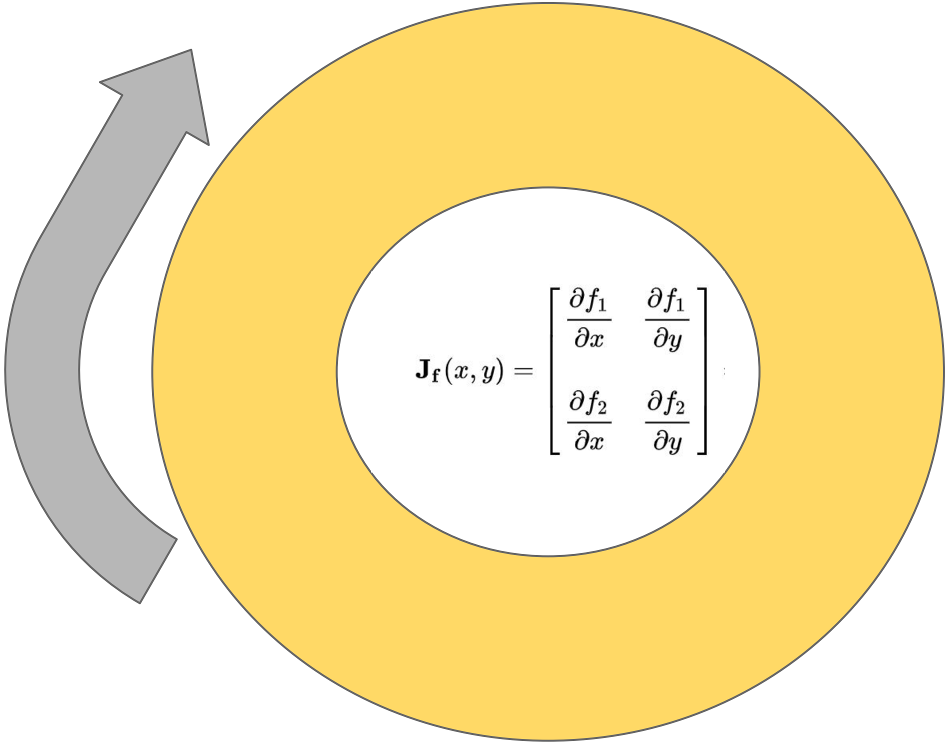


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00

Task 5

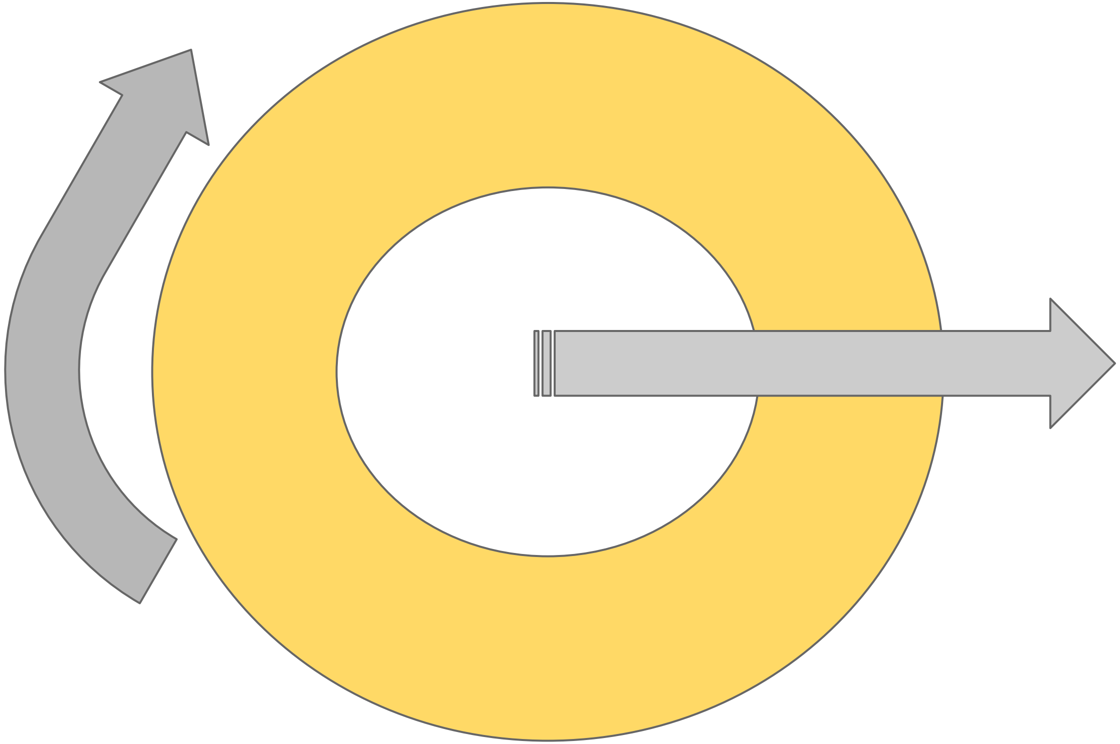
Write the method __SimplifiedNewton__ which will compute the Jacobian only once.

```
for i in range(1000000):
```



$$\mathbf{J_f}(x,y) = \begin{bmatrix} \dfrac{\partial f_1}{\partial x} & \dfrac{\partial f_1}{\partial y} \\[2ex] \dfrac{\partial f_2}{\partial x} & \dfrac{\partial f_2}{\partial y} \end{bmatrix}$$

```python
for i in range(1000000):
```



$$\mathbf{J_f}(x, y) = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{bmatrix}$$

Comparison of equations

- Newton's method
- Simplified Newton's method

$$J_F(x_n)(x_{n+1} - x_n) = -F(x_n)$$

$$J_F(x_0)(x_{n+1} - x_n) = -F(x_n)$$

Task 5

- We made a method __SimplifiedNewton__
- It is almost an exact copy of the original Newton Method
- The Jacobian is calculated outside of the for-loop (saving computer power)
- A **Boolean parameter** now allows us to choose which of the methods we want to use when plotting

Task 6

Compute the derivatives numerically for the Jacobian and add these derivatives as an optional argument in __init__.

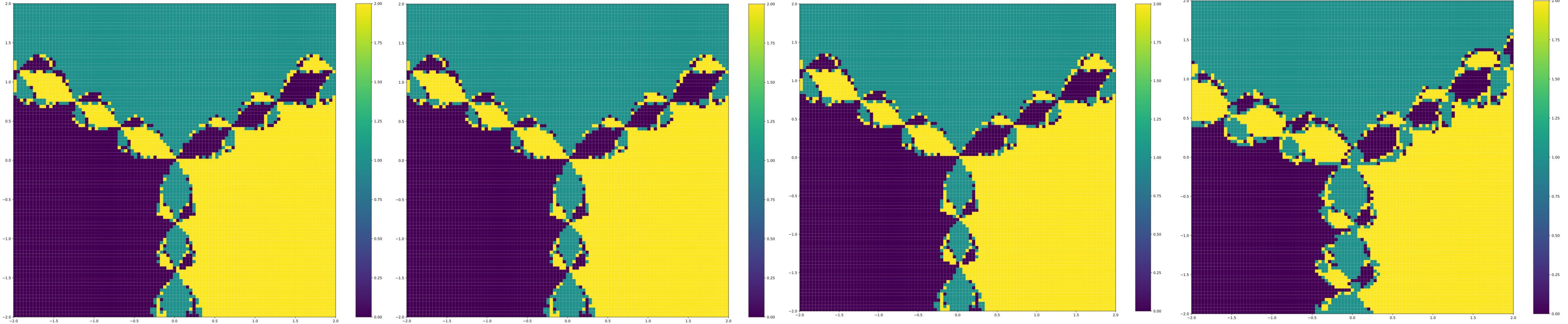
Task 6

- Set up partial derivatives inside the Jacobian

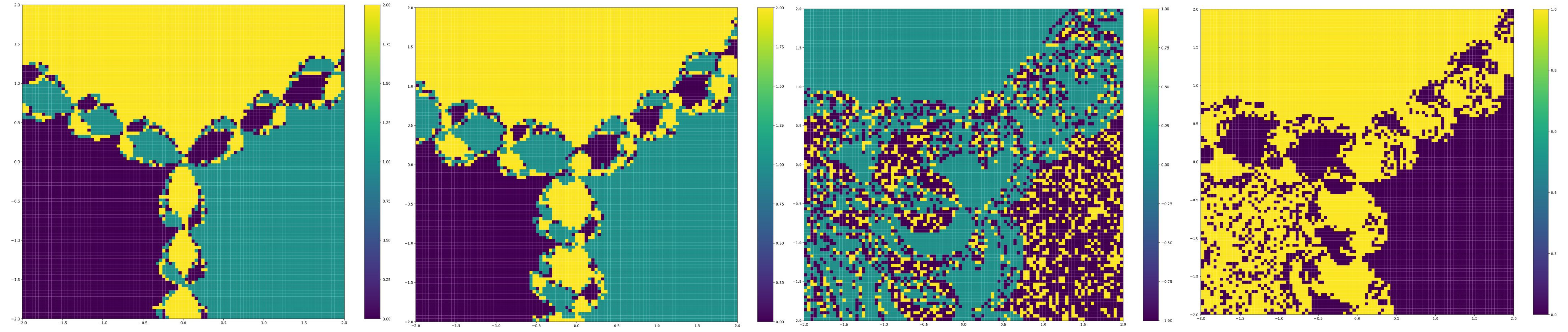
$$\frac{\partial f}{\partial x_i}(a_1,\ldots,a_n) = \lim_{h\to 0} \frac{f(a_1,\ldots,a_i+h,\ldots,a_n) - f(a_1,\ldots,a_i,\ldots,a_n)}{h}.$$

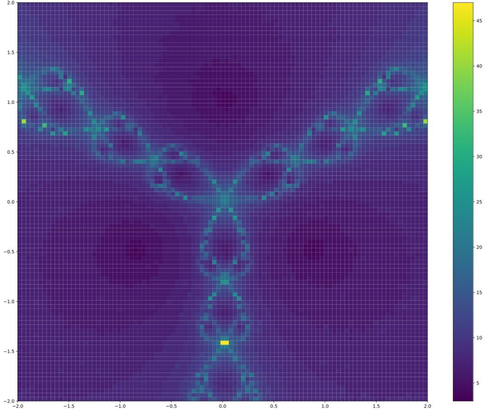
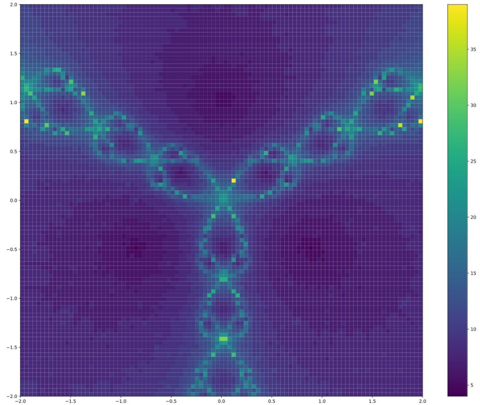
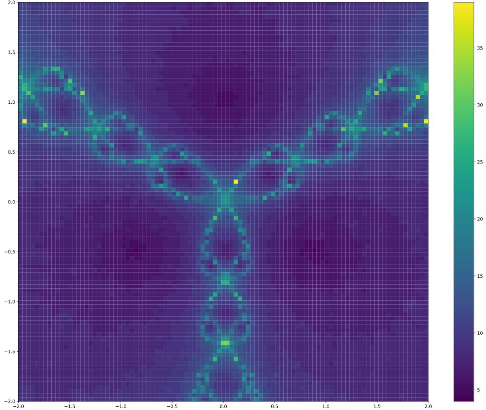
- Modified the code so that the derivative argument of the __init__ method is optional

- Tested different values of h to see if this affected the plot

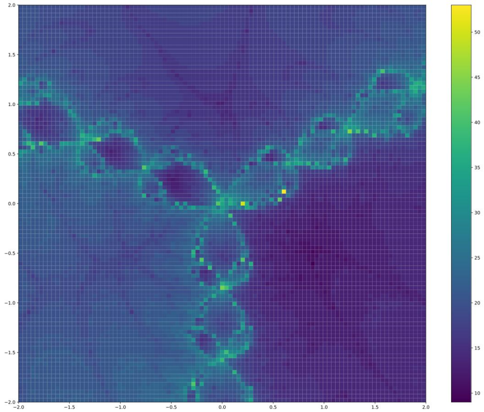
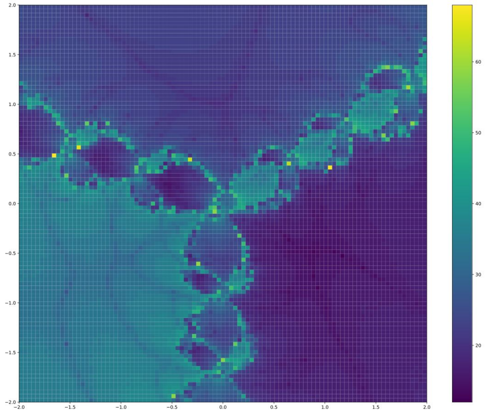
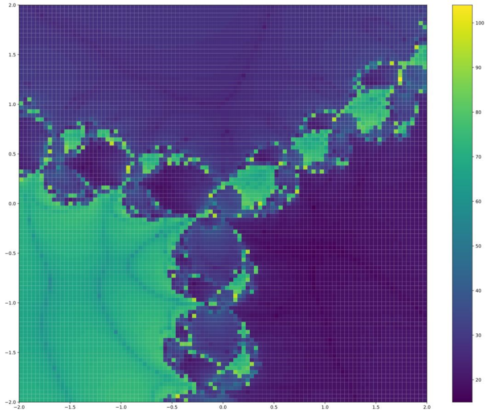


Plot using increasing values for h



ItPlot more iterations needed for convergence

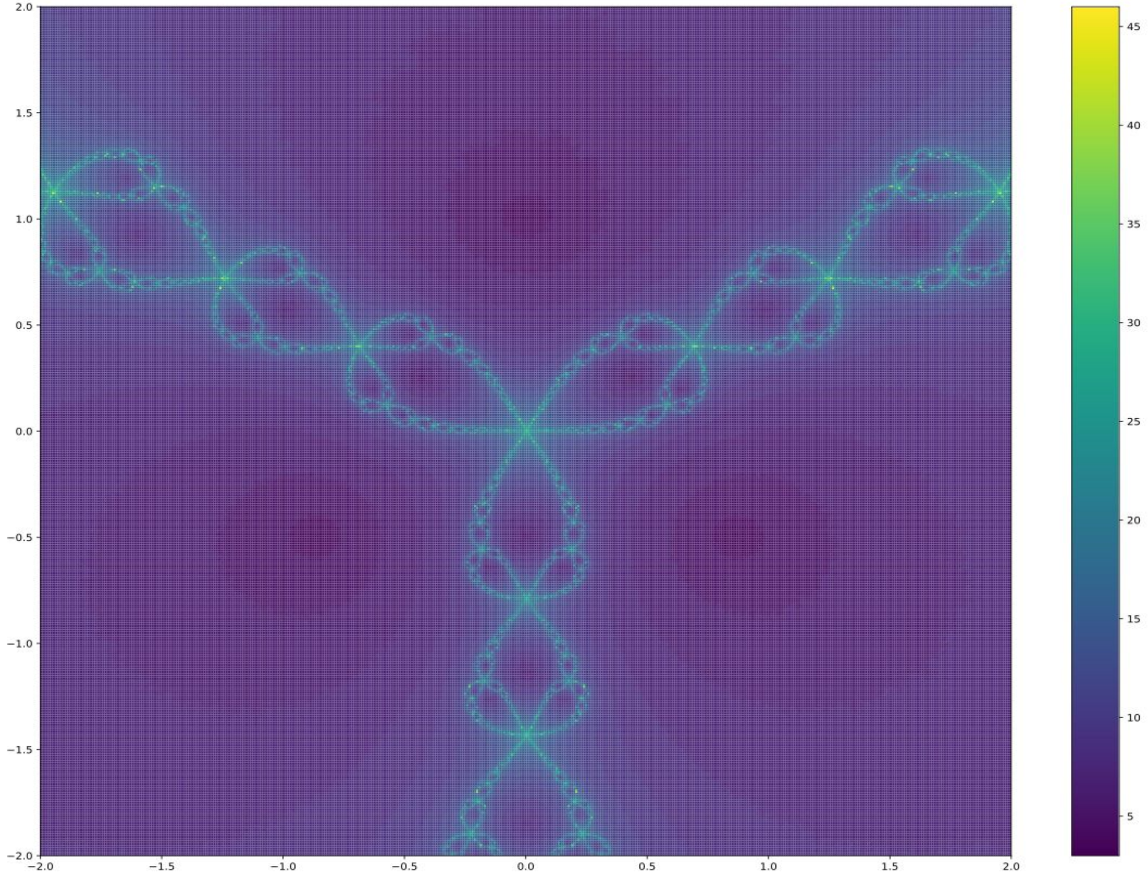
Task 7

Write the method __iPlot__ which show dependence between initial values and iterations needed to reach convergence.

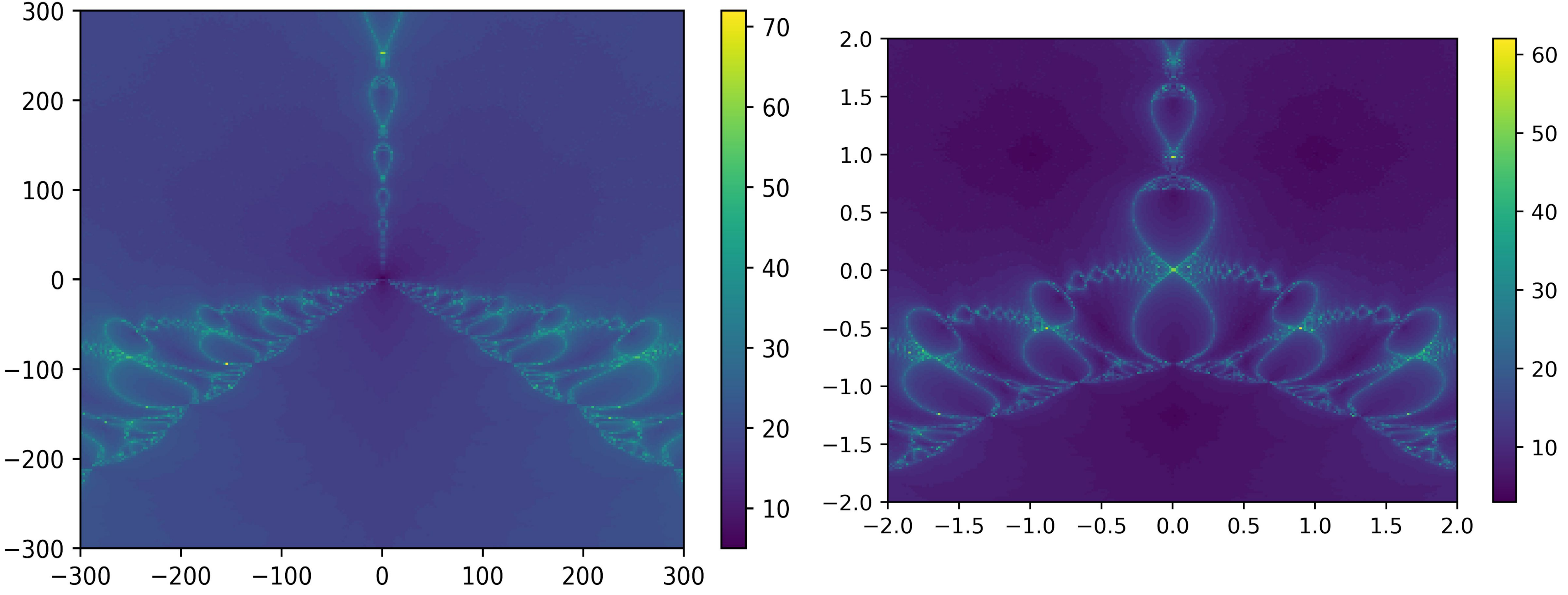
Task 7

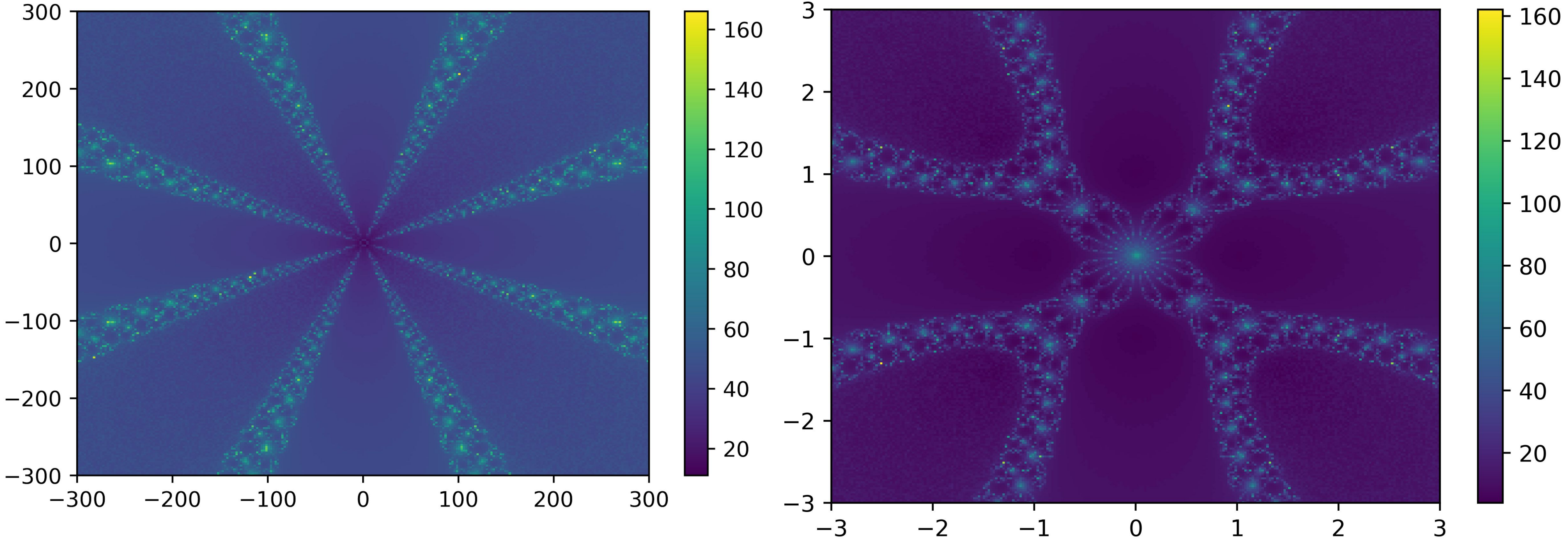
- In Newton-Raphson method from an initial value reasonably close to the actual root of a given equation. It can approximate by the intersection of its tangent line until reach the actual root.

- The __itPlot__ method displays the numbers of iterations in X and Y-axis depending on a range of initial guesses, in this case [-2,2]. Where the colours are brightest we have the most iterations necessary for convergence.



Starting values plotted against number of iterations


300
200
100
0
−100
−200
−300
−300 −200 −100 0 100 200 300
70
60
50
40
30
20
10
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0
60
50
40
30
20
10



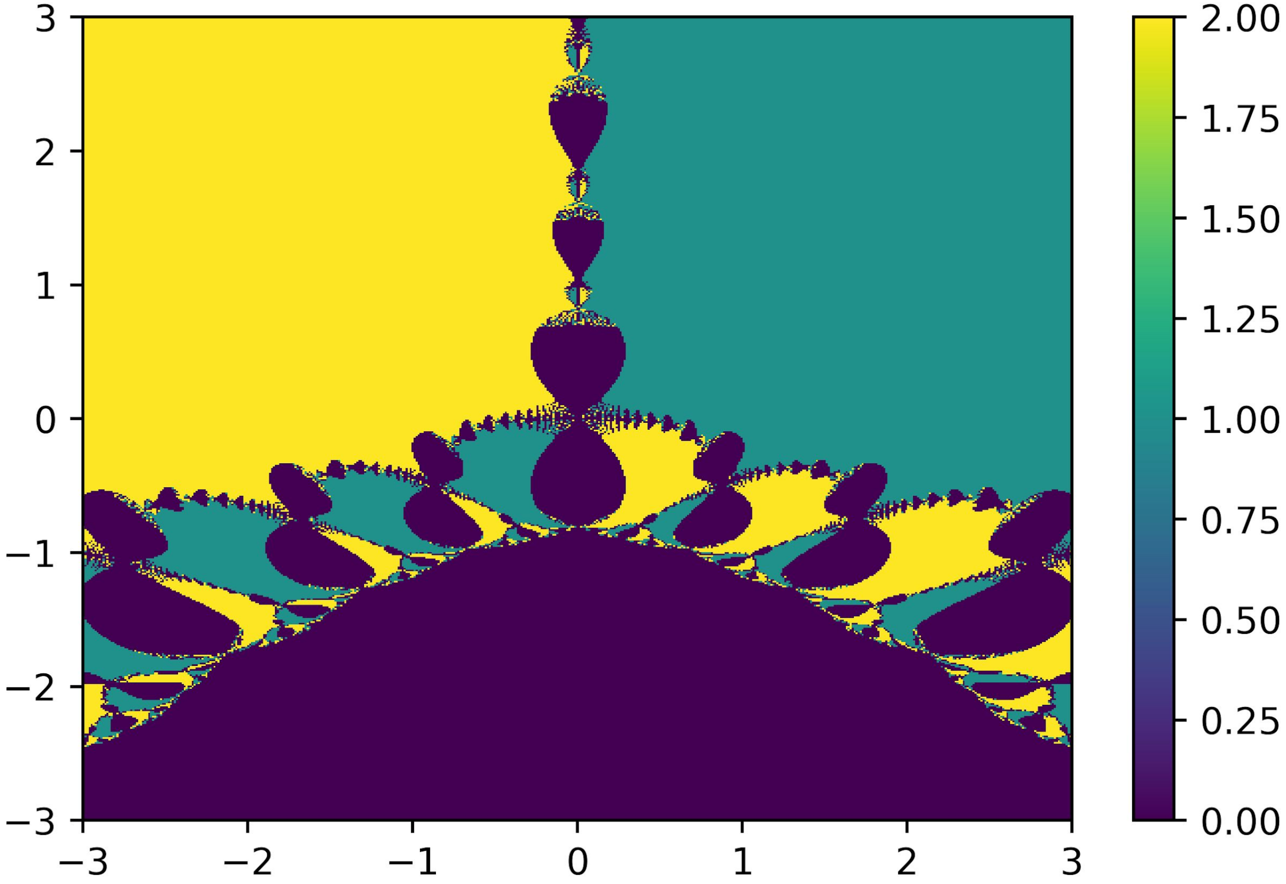
300
200
100
0
−100
−200
−300
−300
−200
−100
0
100
200
300
160
140
120
100
80
60
40
20
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3
160
140
120
100
80
60
40
20

Task 8

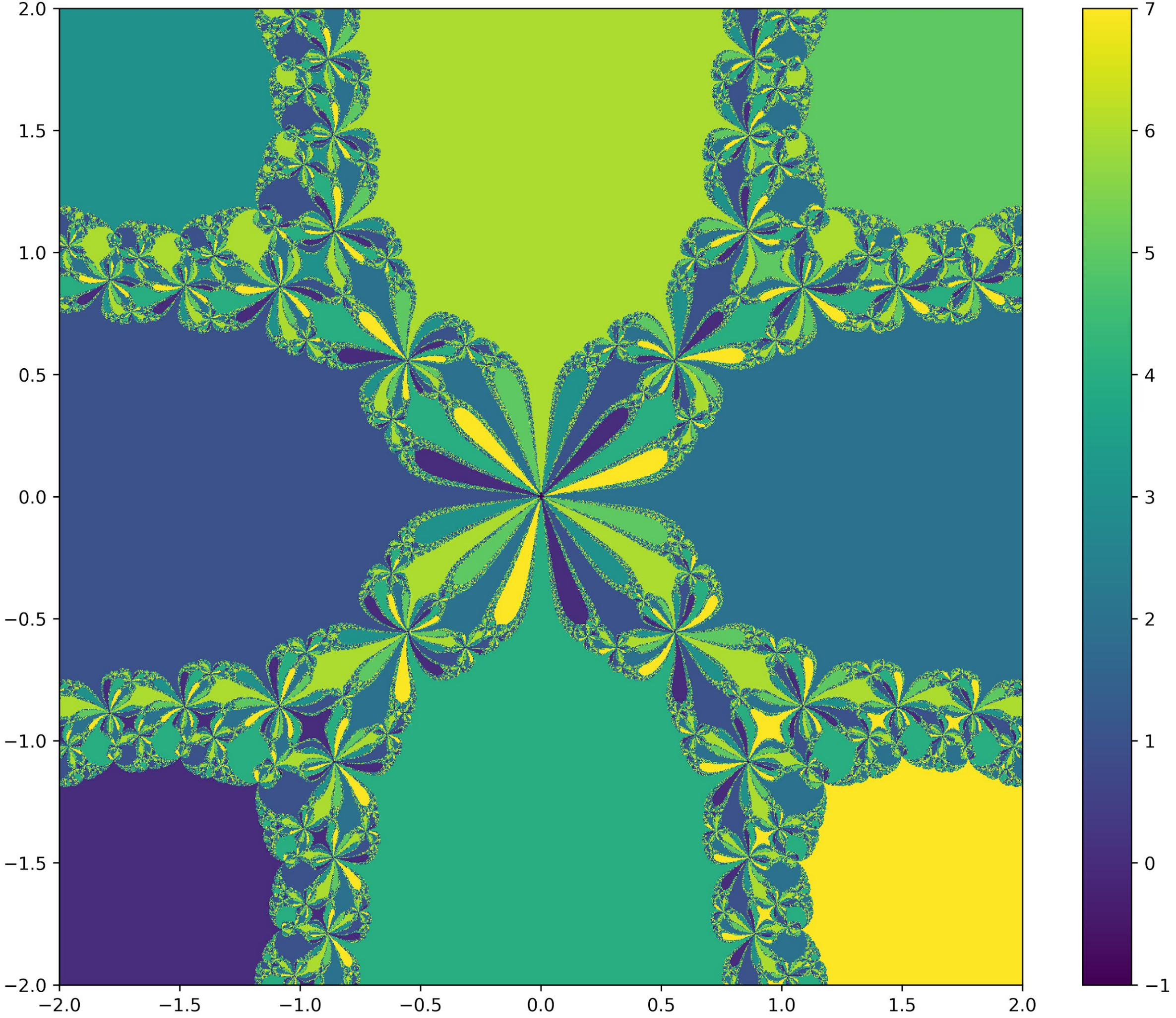
Trying the code with these given functions

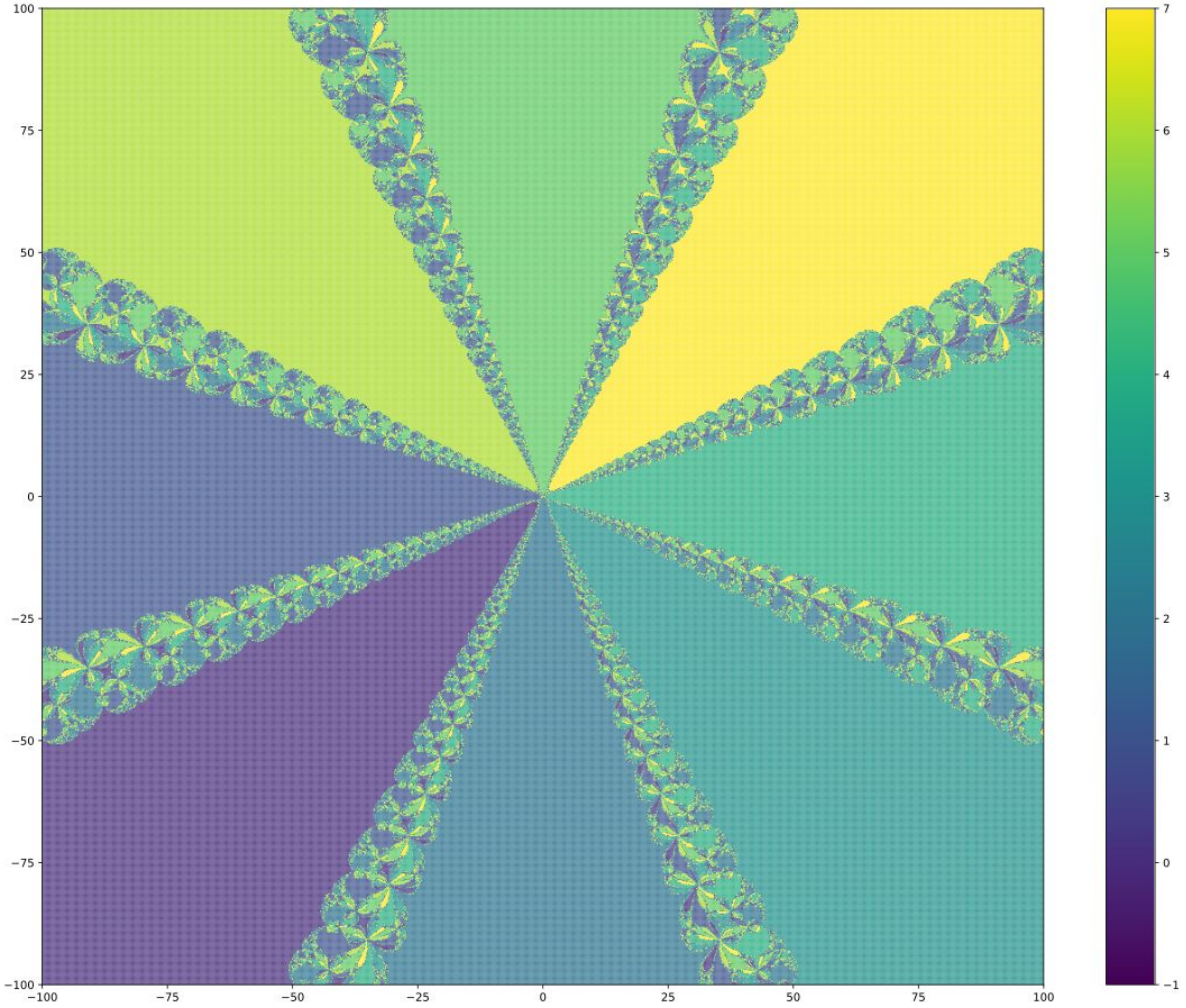
$$F(x) = \begin{pmatrix} x_1^3 - 3x_1x_2^2 - 2x_1 - 2 \\ 3x_1^2x_2 - x_2^3 - 2x_2 \end{pmatrix}$$

$$F(x) = \begin{pmatrix} x_1^8 - 28x_1^6x_2^2 + 70x_1^4x_2^4 + 15x_1^4 - 28x_1^2x_2^6 - 90x_1^2x_2^2 + x_2^8 + 15x_2^4 - 16 \\ 8x_1^7x_2 - 56x_1^5x_2^3 + 56x_1^3x_2^5 + 60x_1^3x_2 - 8x_1x_2^7 - 60x_1x_2^3 \end{pmatrix}$$



3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
7
6
5
4
3
2
1
0
−1



100
75
50
25
0
−25
−50
−75
−100
−100
−75
−50
−25
0
25
50
75
100
7
6
5
4
3
2
1
0
−1

```python
    def __Newton__(self,x0):
        xt= np.array([x0[0],x0[1]])
        f,g,tol= self.f,self.g,self.tol

        for i in range(200):
            prev=xt #row vec
            fvec = np.array([f(xt),g(xt)])
            J = self.__Jacobian__(prev)
            xt= xt - np.linalg.solve(J,fvec) # col vec
            if abs(xt-prev).all() < tol:
                return xt,i
        else:
            return "No conv detected"
```

Numerical

```python
def __Jacobian__ (self, xvec):
    f, g = self.f, self.g
    fvec = np.array([f,g])
    xvec=list(xvec)
    J = np.zeros([len(fvec),len(xvec)])
    h = 1.e-8
    for i in range(len(fvec)):
        for j in range(len(xvec)):
            xvech = xvec.copy()
            xvech[j] += h
            J[i,j] = (fvec[i](xvech) - fvec[i](xvec))/h
    return J
```

Symbolic

```python
def __Jacobian__ (self, xvec):
    f, g = self.f, self.g
    fvec = np.array([f,g])
    xvec=list(xvec)
    J = np.zeros([len(fvec),len(xvec)])
    xl = symbols('x0 x1')
    Mw=np.array([[diff(f(xl), x0),diff(f(xl), x1)],
        [diff(g(xl), x0),diff(g(xl), x1)]])
    for i in range(2):
        for j in range(2):
            Jsym = diff(fvec[i](xl),xl[j])
            J[i,j] = Jsym.subs([(xl[0],x[0]),(xl[1],x[1])])
    return J
```

```python

    def __getzeroes__(self, xinitial):
        N = self.__Newton__(xinitial)[0]
        if self.listempty==True:
            self.xz.append(N)
            self.listempty=False
        for i in range(len(self.xz)):
            if type(N)== str:
                return -1
            C=abs(self.xz[i]-N)<1.e-5
            if C.all()==True:
                break
        else:
            self.xz.append(N)
        return i
```