ECE 122: Introduction to Programming for ECE- Spring 2021

Project 4: Introduction to Scientific Computing (Numpy, Matplotlib)

Due Date: **See website, class policy and moodle for submission**
This project can be done alone or by team of two students (max). For team of two, use only one submission on moodle, add your two names in comment at the top of the `pi.py` file.

Description

The goal of this project is for you to get some exposure to scientific computing and practice various numerical computations using Numpy:

The project must include several files:

1. `pi.py`, `triangle.py`, `square_root.py`, `newton_fractal.py` and `decoder.py`: the application files that you need to implement.
2. `matrix2image.py`, `lion.txt`, `secret.coo`: files (provided) for task5, and `secret.png` that you must generate.

Do not forget to comment your code. Make sure you obtain the **exact same output** for the **exact same input** for the application examples. Your program will be tested with different input by the graders.

Submission/Grading Proposal

You will regroup all your files into one zip file at the time of submission. This project will be graded out of 100 points:

1. Your program should implement all basic functionality/Tasks and run correctly (100 points).
2. The overall programming style grade is included in the 100 grading. It means that up to 10pts will be withdrawn if programs do not have proper identification, and comments.

Task-1- Calculating π using random numbers [30pts]

In mathematics, **Monte-Carlo integration** is a technique for numerical integration using random numbers. It belongs to the class of stochastic methods. Unlike deterministic methods that evaluate a definite integral using a regular grid (think about the Riemann sum what you have seen in calculus), Monte Carlo randomly choose points at which the integrand is evaluated. The evaluation of the integral would converge when the number of random sample increases. This method is particularly useful to evaluate higher-dimensional integrals in multi-variable calculus.

Let us assume that you want to evaluate the following integral:

$$v = \int_{\Omega} f(\mathbf{x}) d\mathbf{x},$$

with Ω representing the entire domain (domain in 1D, area in 2D or volume in 3D) where the integral of the function f must be evaluated, and $\mathbf{x}$ represents the function variables (x in 1D; (x,y) in 2D; (x,y,z) in 3D). Remark: you can also go beyond dimension 3!

Using the Monte-Carlo algorithm, this integral can be approximated by:

$$v \simeq \frac{|\Omega|}{n} \sum_{i=1}^{n} f(\mathbf{x}_i),$$

where n represents the number of points $\mathbf{x}_i$ (samples) chosen randomly in Ω. At the limit of large n (law of large numbers), this expression will converge to the solution v.

For Task1, let us consider the example of the circle of radius 1. It means that the area of this circle should be equal to π. We propose then to calculate π by evaluating the area of the circle using Monte Carlo integration. It is common practice to consider a domain (area in this case) Ω to be a square of dimension [-1:1] for x and [-1:1] for y, so it will enclose the circle. We note that Ω covers a total area equal to 4 ($|\Omega| = 4$). A random point i chosen in Ω will have the coordinate (x_i, y_i). To represent the circle (and make sure that $v = \pi$), the function $f(x_i, y_i)$ ($\mathbf{x}_i = (x_i, y_i)$ in 2D) must satisfy:

$$f(x_i, y_i) = \begin{cases} 1 & \text{if} \quad x_i^2 + y_i^2 \leq 1 \\ 0 & \text{else} \end{cases},$$

it means that f will be equal to 1 if a random point is selected within the circle (or 0 otherwise). By applying the formula above, it comes that:

$$\pi \simeq \frac{4}{n} \sum_{i=1}^{n} f(x_i, y_i).$$

We propose to evaluate this expression for all n going from 1 to $1,000,000$ and see how good the approximation of π becomes. When running your `pi.py`, you should get:

```
Using 10 samples, pi is 3.2
Using 100 samples, pi is 2.88
Using 1000 samples, pi is 3.14
Using 10000 samples, pi is 3.1444
Using 100000 samples, pi is 3.15004
Using 1000000 samples, pi is 3.141336
```

In addition, a couple of Figures would appear. Figure 1 (left) shows the whole variation of π with the number of samples. It is using a log scale for the x-axis. Figure 1 (right) represents the position of the random points for four different values of n.

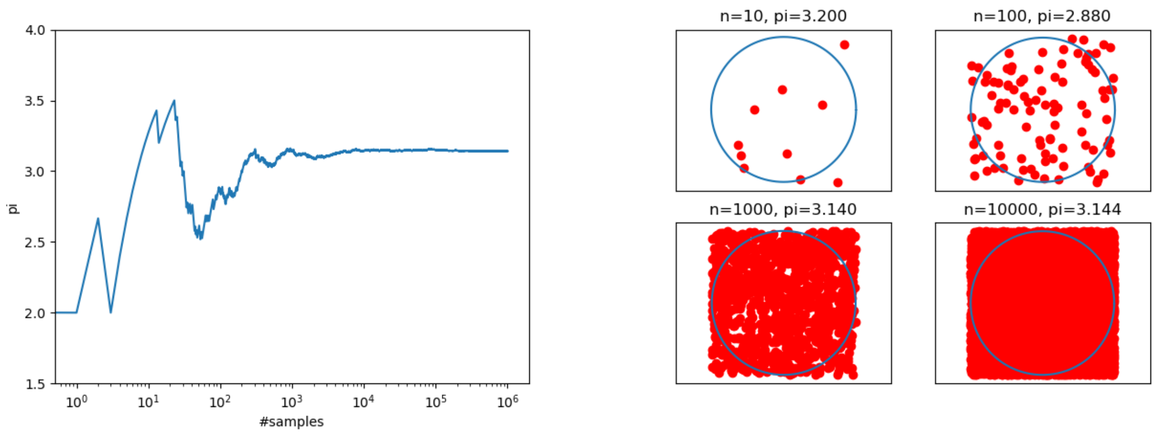

Figure 1: (Left) Variation of the calculated value of π along the number of random samples. (Right) Various positioning of random sample points.

How to proceed:

1. Using Numpy random, generate 1 million uniform random numbers along the x axis, and along the y-axis. This would correspond to 1 million random coordinates (x,y). At the same time that you are counting how many of these points lies within the circle, you can calculate the current approximate value of π by the formula above (all these running π approximate values can be stored in a Numpy array of size n). Example using n=3 (to give you an idea):

```
    *Generate 3 random values for x and y
    *Compute d=sqrt(x^2+y^2) we get [d0,d1,d2]
    *Initialize pi=[0,0,0] and m=0
    *check if d0 inside circle d0<=1 (here, suppose it is)
    *m=m+1 (m=1)
    *pi[0]=(4/1)*m=4
    *check if d1 inside circle d1<=1 (here, suppose it is not)
    *pi[1]=(4/2)*m=2
    *check if d2 inside circle d2<=1 (here, suppose  it is)
    *m=m+1  (m=2)
    *pi[2]=(4/3)*m=8/3=2.666
```

2. Display results on screen (to get the same output than shown above). Your solution should be fast. Your figure/result should appear almost instantaneously.

3. Use a random seed of 7 to reproduce the results show here (figure and output).

4. Using pyplot, plot the two figures. You can use the instructions `plt.figure(0)` and `plt.figure(1)` to switch between figure plots. For the first figure, you can use the method `plt.ylim` to set the y-range between [1.5,4]. In the second figure, π is displayed using 3 decimal digit accuracy, this can be done using the format %.3f (in place of %s for example). You can also get rid of ticks and label marks using `plt.xticks([])` and `plt.yticks([])`. In order to reproduce the figure you also need to use the `plt.scatter` method rather than `plt.plot` (look it up).

Task-2- Calculating area of triangles using random numbers [30pts]

Let us consider another example for the Monte-Carlo integration. Here the user is asked to enter the (x,y) coordinates of 3 points (corresponding to the 3 corners of a triangle) that we will call x_1, y_1 for point 1, x_2, y_2 for point 2 and x_3, y_3 for point 3. Alternatively, the user can also choose the default values that must be hard-coded. Similarly to the circle example, the program will display various evaluations about the area of the triangle. For this Task, we will use only $n = 100,000$ random samples.

This is an example of execution of `triangle.py`:

```
Enter (x,y) of  point-1, default is (0.5,0.5): 0.5 0.5
Enter (x,y) of  point-2, default is (3,2.5):
Enter (x,y) of  point-3, default is (1,3): 1 3
Barycentric Matrix
[[0.5 3.  1. ]
 [0.5 2.5 3. ]
 [1.  1.  1. ]]
Using 10 samples, area of triangle is 1.25
Using 100 samples, area of triangle is 2.375
Using 1000 samples, area of triangle is 2.6375
Using 10000 samples, area of triangle is 2.61
Using 100000 samples, area of triangle is 2.64075
```

The default coordinates are also used in this example (for point-1 and point-3, coordinates were reentered just as an example, for point-2 just Enter was pressed). Again similarly to the circle case, two output figures are plotted which are given in Figure 2 (Left) and Figure 2 (Right).

To compute these results we have chosen a domain Ω to be a square of dimension $[x_{min} = \min(x_1, x_2, x_3){:}x_{max} = \max(x_1, x_2, x_3)]$ for x and $[y_{min} = \min(y_1, y_2, y_3){:}y_{max} = \max(y_1, y_2, y_3)]$ for y, so it will enclose the triangle. We note that Ω covers a total area ($|\Omega|$) equal to $(y_{max} - y_{min}) * (x_{max} - x_{min})$. To represent the integration on the triangle, the function $f(x_i, y_i)$ must now satisfy:

$$f(x_i, y_i) = \begin{cases} 1 & \text{if} \quad (x_i, y_i) \quad \text{belongs to the triangle} \\ 0 & \text{otherwise} \end{cases},$$

So how do we determine if a random point of coordinate (x_i, y_i) lies inside the triangle? The barycentric coordinate system tells us that any point (x, y) can be obtained using the coordinates of the vertices of a triangle via linear combinations of these coordinates. The sum of the coefficients of the linear combinations should also be equal to 1. In 2D, we get:

$$\begin{cases} a_1x_1 + a_2x_2 + a_3x_3 = x \\ a_1y_1 + a_2y_2 + a_3y_3 = y \\ a_1 + a_2 + a_3 = 1 \end{cases}$$

We note that x_1,x_2,x_3 are the x-coordinates of the triangle vertices, y_1,y_2,y_3 are the y-coordinates of the triangle vertices, x,y represents the coordinate of a given point (our random point), and a_1,a_2,a_3 are the coefficient of the linear combination that must be determined. The expression above is then a system of linear equations with 3 unknowns a_1,a_2,a_3. In high-school math, you have learned how to solve such system by substitution or elimination. You could do the same here and get a formula for a_1,a_2,a_3 in function of all other known quantities. Another approach is to use a matrix system (linear algebra), the system matrix associated to the equations above is given by:

$$\begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

In this case, you could use the solve function offers by Numpy to obtain the solutions a_1,a_2,a_3 (as briefly seen in class notes).

Once a_1,a_2,a_3 obtained, the barycentric approach tells us that if all these coefficients are greater than 0, then the point (x, y) must be located inside the triangle!

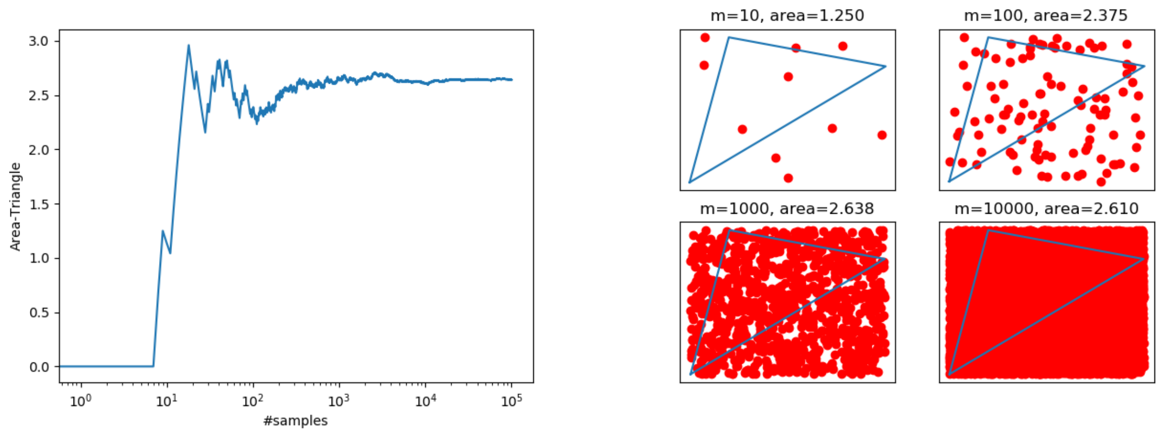

Figure 2: (Left) Variation of the calculated area of the triangle along the number of random samples. (Right) Various positioning of random sample points.

How to proceed:

1. Once the input data are entered and the boundary of the domain Ω are calculated, this is actually very similar to Task1, where you also need to generate the x and y random coordinates. Again use 7 for random seed. Here we consider $n = 100,000$ only.

2. Again, you need to count how many of these random numbers are within the triangle. You will need to calculate the coefficients a_1,a_2,a_3 of each random point by solving the linear system. The matrix can be generated only once at the beginning of the simulation. At the same time you are counting the points inside the triangle, you can calculate the approximate values of the area of the triangle given by the integration formula.

3. Display results on screen (to get the same output than shown above). This includes the barycentric matrix.

4. Using pyplot, plot the two figures. The triangle itself can be plotted using a simple trick. Define a couple of arrays xt, yt of dimension 4, the first three elements would contain the x,y coordinates of the three triangle vertices (corner points) while the last one would contain (duplicate) the coordinate of the the first vertex (pyplot will consider that the loop is closing).

Task3- Calculating square roots using Newton iterations- [10pts]

The Newton method also called the Newton-Raphson iteration method, is a root-finding algorithm. It means that it will find the value x such that $f(x) = 0$ where f is a given function. It proceeds by successive iterations until convergence (forming a series). Let us start with an initial guess x_0, the new iterate x_1 (that supposes to be closer to the true solution x that we are looking for) is given by:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

where $f'(x_0)$ is the derivative of $f(x)$ evaluated at x_0. We can can continue this process for a certain number of iteration or until the calculated solution starts to converge (i.e x_{i+1} at step $i+1$ is close to x_i):

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

This method has a very large number of applications. As an example let us suppose that we want to solve:

$$x^2 = 5,$$

as we know the result is $x = \sqrt{5}$ (could be positive or negative). But...how do we compute square root in practice?
The equation above can also be understood as finding x such that $f(x) = x^2 - 5 = 0$ (so finding the roots of x). Let us then use the Newton iteration here, it comes at iteration $i + 1$:

$$x_{i+1} = x_i - \frac{(x_i^2 - 5)}{2x_i}.$$

Let us start this process using $x_1 = 2$, we get:

$$\begin{aligned} x_1 &= 2 \\ x_2 &= 2.25 \\ x_3 &= 2.236111111111111 \\ x_4 &= 2.236067977915804 \\ x_5 &= 2.236067977499789 \\ x_6 &= 2.236067977499789 \end{aligned}$$

which is actually converging to $\sqrt{5}$. If we start the process with a point closer to -$\sqrt{5}$ then it will converge to this other root as well.

Using the Newton iteration approach, implement the program `square_root.py` that should execute as follows (example of 3 executions here):

```
Square root of which number? 3
1 2.0
2 1.75
3 1.7321428571428572
4 1.7320508100147276
5 1.7320508075688772
```

```
Square root of which number? 789.5
1 395.25
2 198.6237349778621
3 101.29929361423272
4 54.546515046921506
5 34.51020015234239
6 28.693747150294474
7 28.10422628102932
8 28.098043316733776
9 28.09804263645424
```

```
Square root of which number? Square root of which number?  25
1 13.0
2 7.461538461538462
3 5.406026962727994
4 5.015247601944898
5 5.000023178253949
6 5.000000000053723
7 5.0
```

Here, we stop the iteration when the relative error with the true solution (that is printed as well) is less or equal to 10^{-15}. Hint: if x_i is the approximation of $\sqrt(a)$ at step i, relative error is $|x_i - \sqrt{a}|/|\sqrt{a}|$.

Task4- Newton Fractal- [20pts]

What is a Newton fractal? Let us suppose that we want to find the solutions of $z^n = 1$ where z is a complex number this time (so $z = x + jy$), and n is a given integer. There are actually n solutions given by (j stands for the imaginary number):

$$\exp j\frac{2\pi m}{n} \quad \text{with} \quad m = 0, 1, .., n-1 \tag{1}$$

.

Remark: You do not need to know complex numbers to complete this task, just think of (x,y) coordinates for the complex plane and follows the steps.

Using the Newton iteration, these solutions can be computed by finding the roots of the following function $f(z) = z^n - 1$. Now if we consider a point z_0 in the complex plane to start the Newton iteration, it may converge to either one of these roots and that may not depend on how close the starting point is to a given root! In some region of the complex plane, two infinitely close points could actually converge to two different roots! The system presents then some chaotic behavior with rather "strange attraction". Let us decide to use a particular color for a particular root, at each time a starting point z_0 converge to a given root, we color the original point z_0 by the corresponding root color. At the end (after scanning the entire complex plane), we will obtain a beautiful fractal!

Here is the output execution of newton_fractal.py:

```
Newton fractal z**n=1, Enter n (default 3):
Enter xmin,xmax,ymin,ymax (default -1.35,1.35,-1.35,1.35):
Solutions are
(1+0j)
(-0.4999999999999998+0.8660254037844387j)
(-0.5000000000000004-0.8660254037844384j)
```

By default, the program will then apply the Newton iteration on the function $z^3 - 1$. The program is then asking for the x and y boundaries, and it is also printing the corresponding true root solutions calculated by the formula (1) above. In addition, the corresponding Newton fractal is plotted as shown in Figure 3.

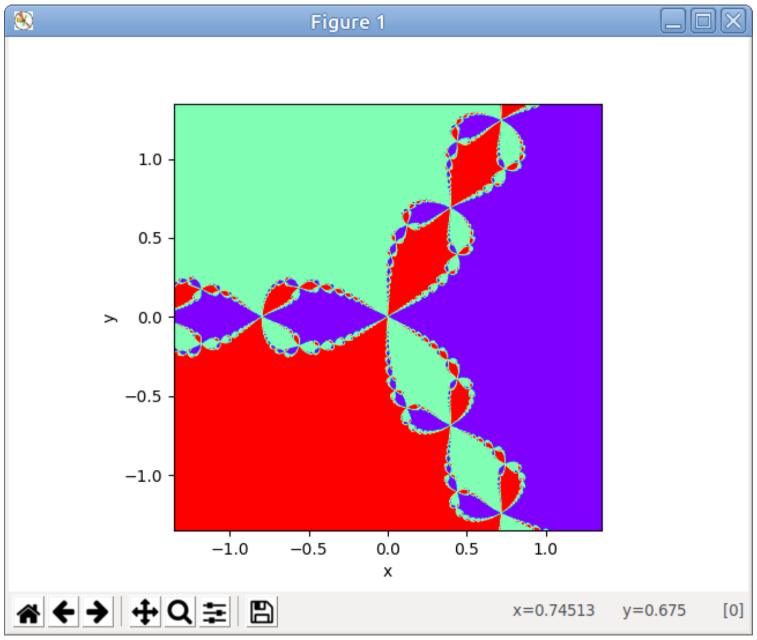

Figure 3: Newton fractal corresponding to $z^3 = 1$.

Let us see another example of execution using a zoom:

```
Newton fractal z**n=1, Enter n (default 3):
Enter xmin,xmax,ymin,ymax (default -1.35,1.35,-1.35,1.35): -0.55 -0.4 0.15 0.24
Solutions are
(1+0j)
(-0.4999999999999998+0.8660254037844387j)
(-0.5000000000000004-0.8660254037844384j)
```

with the corresponding Newton fractal shown in Figure 4.

And again, another example:

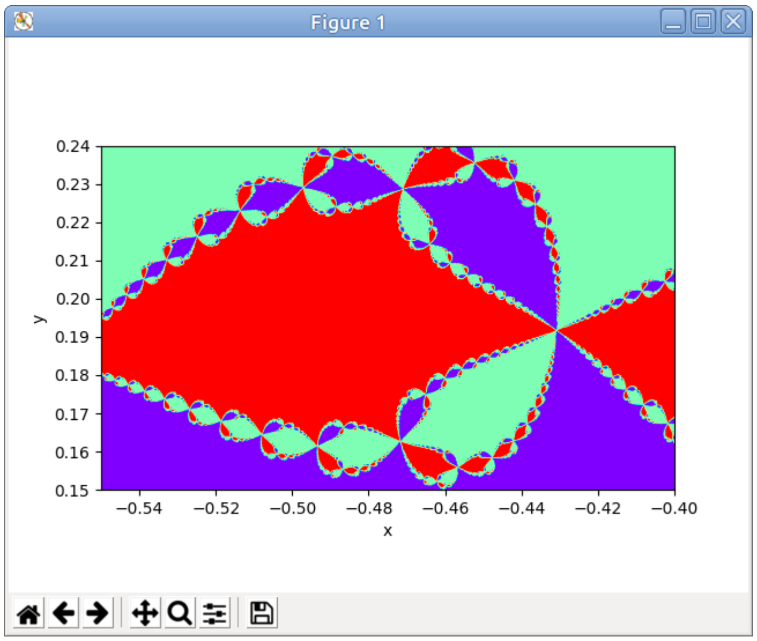

Figure 4: Newton fractal corresponding to $z^3 = 1$, region [-0.55,-0.4,0.15,0.24].

```
Newton fractal z**n=1, Enter n (default 3): 6
Enter xmin,xmax,ymin,ymax (default -1.35,1.35,-1.35,1.35):
Solutions are
(1+0j)
(0.5000000000000001+0.8660254037844386j)
(-0.4999999999999998+0.8660254037844387j)
(-1+1.2246467991473532e-16j)
(-0.5000000000000004-0.8660254037844384j)
(0.5000000000000001-0.8660254037844386j)
```

with the corresponding Newton fractal shown in Figure 5.

How to proceed:

1. Once all the inputs are entered, you will calculate and display the true solutions given in equation (1). In Python, the multiplication by the imaginary number j can simply be done using $1j*$. We note that `exp` and `pi` are available in Numpy (do not import the math module). The solutions can be a stored in a list or a Numpy array (let us call it `sol`).

2. Using the `linspace` method you can define a array of 1000 values for x (from xmin to xmax) and 1000 values for y (from ymin to ymax). In practice, you will choose the minima to be xmin+0.00011 and ymin+0.00011 to avoid getting a value 0 for x and 0 for y (that will cause troubles in our version of Newton iterations). The grid of all possible complex numbers that we will consider, can be computed as a matrix C of size 1000x1000 where each entry is equal to `C[i,j]=x[j]+1j*y[999-i]` and where C has been initialized as

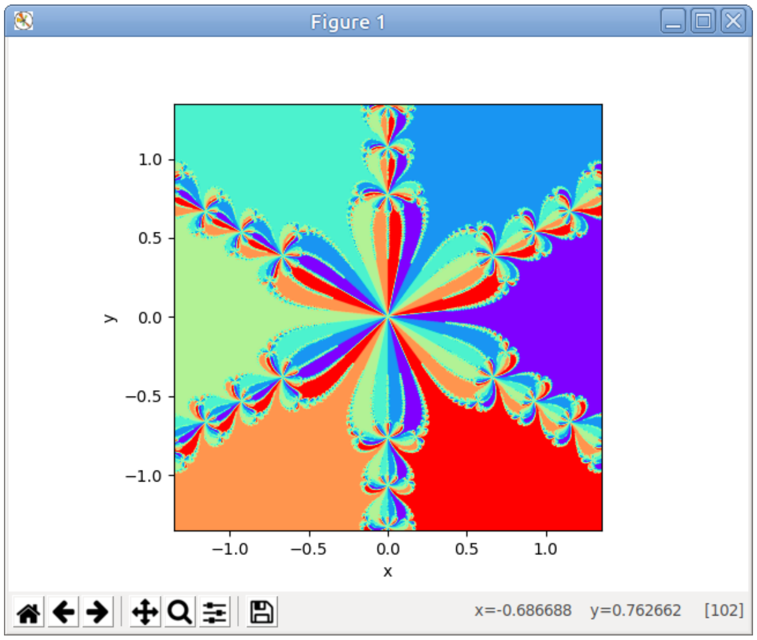

Figure 5: Newton fractal corresponding to $z^6 = 1$.

`C=np.zeros((1000,1000),dtype=complex)`. Alternatively, you can also use the `meshgrid` method (look it up).

3. The vectorization and overloading properties offered by Numpy will allow you to perform the Newton iterations directly using the matrix C (no need of indexing or understand how complex number operations work). Hint: The derivative of `C**n` is `n*C**(n-1)`, also as a reminder: `C**n` means that the power is applied to all matrix elements.

4. You will also use a fix number of Newton iterations equal to 20. Once your iterations done, you can scan over all the elements of the matrix and identify which roots the elements of the matrix are the closest to. One possible option is to check the relative difference between a given element at position `[i,j]` in the C matrix, with all the root solutions i.e. `abs(C[i,j]-sol[m])/abs(sol[m])` for all $m = 0, 1, ..n - 1$, and then keeping track of the index m for which this error is the smallest. Once found, and let us suppose that you have already assigned a corresponding integer between 0 to 255 (representing a color number) to a given root (for example the number `m*255/(n-1)` for the m^{th} root), you can start filling up an integer matrix (of the same shape than C) with this particular color number at the `[i,j]` element position.

5. Your color matrix is now ready to be plotted. Rather than using the `plot` function, to plot a matrix you may want to use the `imshow` function (look it up). To reproduce the results shown here, you need to consider the following function arguments: `cmap,origin,extent,interpolation` that are initialized with values that you need to specify. Note: The `origin` field should be set to `‘‘lower’’`. The `cmap` represents the colormapping, I am using the `‘‘rainbow’’` value but your are welcome to use any mapping you like[1]. The value for the `‘‘interpolation’’` that I am using (to improve the quality of the image) is `‘‘bilinear’’`, again you can use anything you like[2].

[1]https://matplotlib.org/tutorials/colors/colormaps.html
[2]https://matplotlib.org/gallery/images_contours_and_fields/interpolation_methods.html

Task5- Image and Matrices [10pts]

An image can have a matrix representation. Indeed, each pixel of the image could correspond to a given color value in a 2D array. In this activity two matrices representing images are included, the first one is lion.txt which is uncompressed. This means that if you open the file, you will see all the element of the matrix (2d array). The activity also includes the program `matrix2image.py` (already completed for you), that will turn this matrix to a grayscale image. You can execute the following:

```
Enter full matrix file name: lion.txt
```

and you will obtain Figure 6.

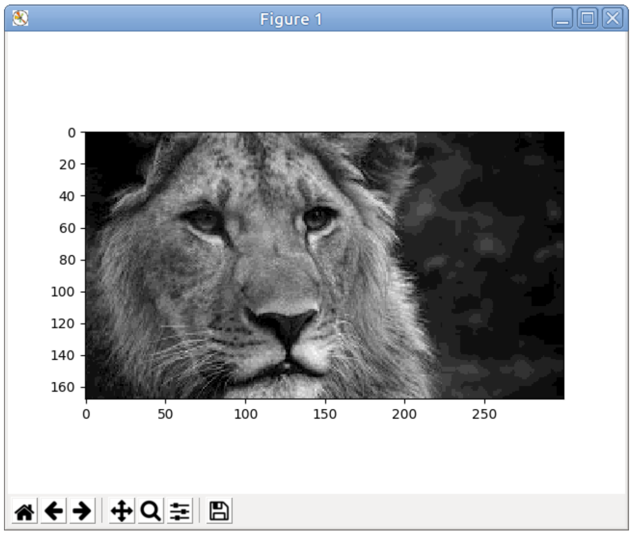

Figure 6: A ferocious lion.

Now, in some situation a matrix may contain a lot of elements equal to zero (in particular if an image contains a lot of white space). To save space in disk, there is no need to store these zeros and one can simply save the coordinates of the non-zero values. If you open the file `secret.coo` (coo stands for coordinate format), you will see that it contains 3 columns, the first columns are the row index of the image matrix (2d array), then the column index, then the corresponding values. All the other elements of the image matrix should be equal to zero. This file represents then a compressed version of an image. But what is this image about? Write the program `decoder.py` that will execute this way:

```
Enter compressed coordinate matrix file name: secret.coo
```

and plot the secret image! (you will include your image `secret.png` in your zip file).