tutorial4_bonus_material_solutions

October 9, 2024

1 Tutorial 4 Bonus Material

For enthusiasts - this material is entirely optional - you do not need to study it

(It will be useful however for Group Project 7 – Recursive Animations and the Towers of Hanoi.)

1.1 Bonus Mini Project: Constructing the Mandelbrot Set

[1]:
```
# RUN THIS CELL FIRST
import numpy as np
import matplotlib.pyplot as plt
```

The **Mandelbrot set** is the set of values c in the complex plane such that the sequence $(z_n)_{n\geq 0}$ defined by the recurrence relation

$$z_0 = 0 \quad \text{and} \quad z_n = z_{n-1}^2 + c, \quad \text{for } n = 1, 2, 3, ...$$

is bounded. We say that $(z_n)_{n\geq 0}$ is the *Mandelbrot sequence induced by* c. (It would be more correct to write it as $(z_n^c)_{n\geq 0}$ to explicitly show the dependence on c. However this would clutter our notation.)

Reminder 1. $(z_n)_{n\geq 0}$ being bounded means that, for some natural number (or positive real number) N, $|z_n| \leq N$ for all $n \geq 0$.

Reminder 2. For a complex number $z = a + bi$, the *conjugate* of z is defined to be $\overline{z} := a - bi$ and the *modulus* of z is defined to be

$$|z| := \sqrt{z \cdot \overline{z}} = \sqrt{a^2 + b^2}\,.$$

Note that this means that, given complex number c the sequence $(z_n)_{n\geq 0}$ induced by c starts as: $z_0 = 0$, $z_1 = c$, $z_2 = c^2 + c$, $z_3 = (c^2 + c)^2 + c$, $z_4 = ((c^2 + c)^2 + c)^2 + c$ etc.

Note. It can be shown that, given c, the Mandelbrot sequence $(z_n)_{n\geq 0}$ induced by c diverges to infinity - i.e. is NOT bounded - precisely in the case when $|z_n| > 2$ for some n. Moreover, in general, when $(z_n)_{n\geq 0}$ does diverge, we are able to find some small n such that $|z_n| > 2$. Thus we can compute a good guess as to whether c is in the Mandelbrot set or not by inspecting some fixed number of initial terms of the Mandelbrot sequence $(z_n)_{n\geq 0}$ induced by c, whatever the value of c. Notice also that if $|c| > 2$ then $|z_1| > 2$ and so the sequence $(z_n)_{n\geq 0}$ induced by c diverges.

Complex Numbers in python. The complex number number $a + bi$ is represented as `a + b*j` in python. I.e. in python the $\sqrt{-1}$ is represented by `1j`.

[2]:
```
# Let's check this.
print("Square root of -1 is", (-1)**0.5)
# A couple of examples
z = 3 + 4j
print("Example 1: z =", z)
print("Conjugate of z is",z.conjugate())
print("Real part of z is",z.real)
print("Imaginary part of z is",z.imag)
print("Modulus of z is ", abs(z))

a, b = -2.1, 5.7
u = a + 5.7*1j
print("Example 2: u =", u)
```

```
Square root of -1 is (6.123233995736766e-17+1j)
Example 1: z = (3+4j)
Conjugate of z is (3-4j)
Real part of z is 3.0
Imaginary part of z is 4.0
Modulus of z is  5.0
Example 2: u = (-2.1+5.7j)
```

[3]:
```
# Aside: we also have ways of formatting complex numbers nicely for printing
im_i = (-1)**0.5
print(f"Square root of -1 is {im_i.real:.2f} + {im_i.imag:.2f}i")
# Or as...
print("Square root of -1 is {0.real:.2f} + {0.imag:.2f}i".format(im_i))
```

```
Square root of -1 is 0.00 + 1.00i
Square root of -1 is 0.00 + 1.00i
```

1.1.1 Part 1: a warm up - plotting Mandelbrot sequences

For each of the complex numbers

$$c_0 = 0.3 + 0.2i, \quad c_1 = 0.39 + 0.16i, \quad c_2 = 0.2555 + 0i$$

compute the first 50 terms of the Mandelbrot sequence $(z_n)_{n\geq 0}$ induced by c_k (for $k = 0, 1, 2$) and plot, in a single figure, the values of the moduli $|z_0|, |z_1|, \ldots |z_{59}|$ for each of these three sequences. Use a logarithmic y axis, and appropriate axis labels, a title and a legend. One of the complex numbers c_0, c_1, c_2 belongs to the Mandelbrot set. Which one? For each of the others you should visually estimate at which iterate number n the induced sequences $(z_n)_{n\geq 0}$ starts to diverge to infinity.

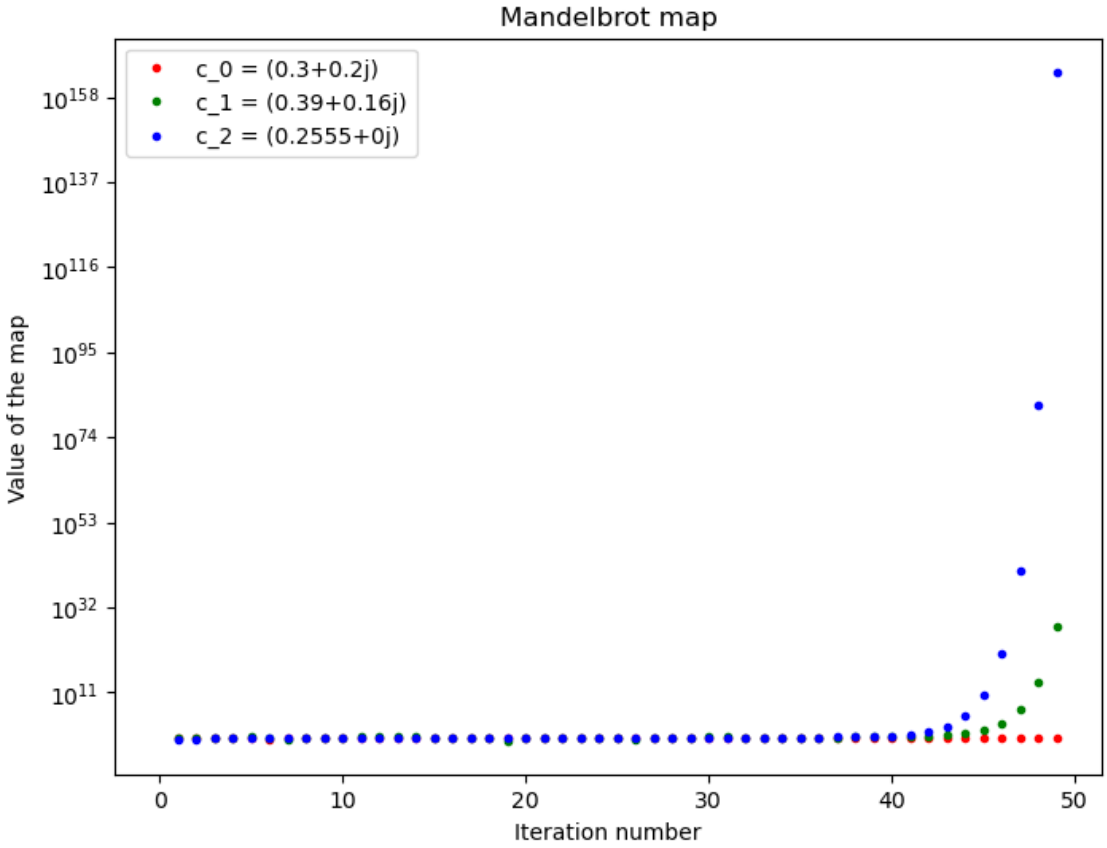
To make your life easier we have written in some preliminary code below. Your task is to write in the necessary code from the point specified by

```
# YOUR CODE HERE
```

onwards.

[4]:
```
# MAX_IT is the number of iterations that we will plot
MAX_IT = 50
# As seen above in python the imaginary "i" is represented by "j"
c_0 = 0.3 + 0.2j
c_1 = 0.39 + 0.16j
c_2 = 0.2555 + 0j

# We put these complex numbers in a list
c_values = [c_0, c_1, c_2]
# A neat way to generate different colors for each plot is to use a␣
 ↪corresponding list
plot_colours = ['r', 'g', 'b']

# Set the figure size
plt.figure(figsize=(8,6))
# z_values is a zero array of length MAX_IT whose components are of complex␣
 ↪number type.
z_values = np.zeros((MAX_IT,), dtype=np.cdouble)

# OVER TO YOU: TWO NESTED FOR LOOPS SUFFICE FOR THE PLOTTING. THE OUTER LOOP␣
 ↪ITERATES OVER
# c_values AND DOES THE NECESSARY PLOTTING. DURING ITERATION k THE INNER LOOP␣
 ↪POPULATES z_values
# WITH THE FIRST MAX_IT MANY VALUES OF THE MANDELBROT SEQUENCE INDUCED BY␣
 ↪COMPLEX NUMBER c_k.
# (THIS IS USED FOR THE PLOTTING) YOU SHOULD THEN SET UP THE Y AXIS AND LABELS␣
 ↪ETC

### BEGIN SOLUTION
# For each k = 0,1,2 we plot the moduli of the first MAX_IT components of the␣
 ↪sequence (z_n) induced by c_k
for k in range(len(c_values)):
    # The inner for loop populates all but the first component of z_values with␣
 ↪ [z_1,...,z_MAX_IT-1] induced by c_k.
    # Note that z_values[0] = 0+0i is unmodifiied (and z_0 is 0+0i by␣
 ↪definition)
    for n in range(1,MAX_IT):
        z_values[n] = z_values[n-1]**2 + c_values[k]
    # For each k, plot the moduli of the array [z_0,z_1,...,z_N-1] induced by␣
 ↪c_k with colour and label
    plt.plot(range(MAX_IT),np.abs(z_values),'.',color=plot_colours[k],␣
 ↪label=f"c_{k} = {c_values[k]}")
```

```
# Make the y-axis logarithmic
plt.yscale('log')

# Decorations
plt.xlabel('Iteration number')
plt.ylabel('Value of the map')
plt.legend()
plt.title('Mandelbrot map')
plt.show()
### END SOLUTION
```



1.1.2 Part 2: the Mandelbrot map

It turns out that, given any complex number c, inspection of the initial segment $(z_0, z_1, \ldots, z_{99})$ comprising the first 100 terms of the Mandelbrot sequence induced by c gives us a good guess as to whether or not c does lie in the Mandelbrot set or not. Here we mean of course that we guess that c is **not** in the Mandelbrot set if and only if for some $n < 100$, $|z_n| > 2$. Moreover when we implement the functions and code that we define below to draw the Mandelbrot set we need to assume a fixed number of iterations throughout our code. Accordingly here we choose to use 100 for this value. However of course we might in future want to refine our work by increasing the number

of iterations. Thus it makes sense to define a global variable assigned to the maximum number of iterations (instead of, for example, passing this in to each function as a parameter). This we now do using the value 100, in the cell below which you should run (otherwise the rest of your code will not execute.)

[5]:
```
MAXITER = 100
```

Now your task is to write a function `mandel` with input parameter `c` which, supposing that that $(z_n)_{n\geq 0}$ denotes the Mandelbrot sequence induced by `c`, should return the least \$n < \$ `MAXITER` such that $|z_n| > 2$ if such an n exists. Otherwise your function should return the number `MAXITER`.

[6]:
```
### BEGIN SOLUTION
def mandel(c):
    # Global variable used: MAXITER
    # YOUR CODE HERE
    n = 0
    z = 0 + 0j
    # Compute z = z_n at each iteration and continue for as long
    # as the sequence (z_n) does not start to diverge and n < MAXITER
    while abs(z) <= 2 and n < MAXITER:
        z = z**2 + c
        n += 1
    # Then return the first n such that we hit the threshold (or else n =␣
 ↪MAXITER)
    return n
### END SOLUTION
```

[7]:
```
# Test your function here. For example, which of the following two complex␣
 ↪numbers
# belong to the Mandelbrot set.
c1 = 0.366 + 0.1j
c2 = 0.367 + 0.1j
print(mandel(c1))
print(mandel(c2))
# Further tests here...
```

```
100
46
```

1.1.3 Part 3: practice example of pixellated plot using `pcolor`

In section "Pixellated and Contour plots using `pcolor` and `contourf`" of Lecture 4.2 we saw an example of a pixellated plot using `pcolor`. As further practice, and to prepare us for plotting the Mandelbrot we use `pcolor` to plot the surface of the function

$$f(x, y) = \sin(x^2 - y^2)\,.$$

You are given code for doing this below. You should read this code carefully before moving on and...

- Make sure that you understand what all the code does. For example do you fully understand the line

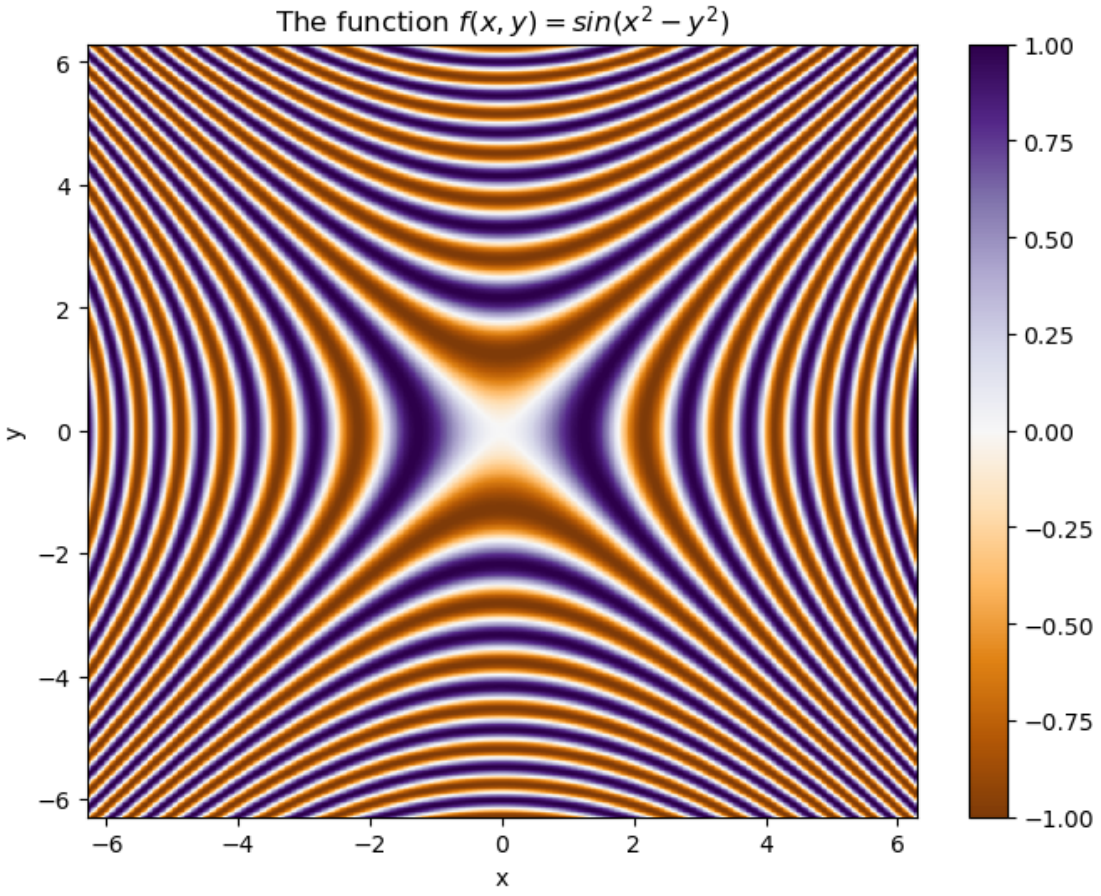
```
x_matrix, y_matrix  = np.meshgrid(x_vector,y_vector)
```

and if not, you should consult Lecture 4.2. - We have plotted over the domain $[-2\pi, 2\pi] \times [-2\pi, 2\pi]$ using space of 0.02 between coordinates. Try plotting over a larger domain, and try varying the space size. - We have used the colormap 'PuOr'. Try using another colour map. Remember that you can find documentation of the colour maps here: https://matplotlib.org/stable/gallery/color/colormap_reference.html .

[8]:
```
# GIVEN EXAMPLE -  COPY THIS CODE INTO THE TESTING AREA
# AND MODIFY IT THERE (SO AS TO PRESERVE THE ORIGINAL HERE)
# RUN THIS CELL TO SEE THE OUTPUT

# Choosing x and y regions. The periodicity is related to  2pi
# We create two 1D arrays over [-2\pi,2\pi] space of 0.02 between each␣
 ↪coordinate.
x_vector = np.arange(-2*np.pi,2*np.pi,0.02)
y_vector = np.arange(-2*np.pi,2*np.pi,0.02)

# To compute the function on a grid of points, we need to turn x_vector and␣
 ↪y_vector into  coordinate matrices
# (See "Generating a 2 dimensional coordinate grid" in Lecture 4_2 for␣
 ↪explanation)
x_matrix, y_matrix  = np.meshgrid(x_vector, y_vector)

# We can then compute the values of f on this grid
f_matrix = np.sin(x_matrix**2 - y_matrix**2)

# By inspection we can see that f should achieve minimum and
# maximum values  -1 and 1 (or very close to these).
f_MIN = f_matrix.min()
f_MAX = f_matrix.max()
print(f"f_MIN = {f_MIN:.6f}, f_MAX = {f_MAX:.6f}\n")

# So let's plot this!
# If you comment out the next line a default size Figure object will still be␣
 ↪created
# by the call to the pyplot pcolor plotting function.
plt.figure(figsize=(8,6))

# We use the pixellated data plotting pyplot function `pcolor`
# We have  chosen the PurpleOrange colormap. For vmin and vmax alternatively
# we can just use -1 and 1 (since we can deduce these by inspection)
plt.pcolor(x_matrix, y_matrix, f_matrix, cmap='PuOr', vmin = f_MIN, vmax =␣
 ↪f_MAX)
```

```
# The colorbar shows us the colour map values
plt.colorbar()
plt.xlabel('x')
plt.ylabel('y')
plt.title("The function $f(x,y)= sin(x^2 - y^2)$")

plt.show()
```

```
f_MIN = -1.000000, f_MAX = 1.000000
```



[9]:
```
# TESTING AREA: COPY THE ABOVE CODE INTO THIS CELL AND
# USE THIS CELL TO MODIFY AND PLAY ABOUT WITH IT AS
# SUGGESTED ABOVE
```

1.1.4 Part 4: a function to create the Mandelbrot set

We now design a function `make_mandelbrot_data` which has six input parameters. Its definition is of the form

```
def make_mandelbrot_data(xmin, xmax, num_xpts, ymin, ymax, num_ypts)
    # Body of the function
```

where the pair `xmin`, `xmax` and the pair `ymin` and `ymax` represent the end points of an interval of x coordinates and an interval of y coordinates respectively (so representing between them a rectangular region of the plane). Also `num_xpts` and `num_ypts` are respectively the number of equally spaced x coordinates and y coordinates that we specify over this segment of the plane. (Often we use the same number of points for both but being able to define these two values separately can be useful for high resolution plots.)

The object of our function is to first create a matrix of x coordinates `x_matrix` and a matrix of y coordinates `y_matrix` representing, as usual (see Lecture 4.2 - "Generating a 2 dimensional coordinate grid"), a grid of two dimensional coordinates of the same dimensions. Here however our coordinates are thought of as being in the complex plane. Our function then creates a matrix of the same dimensions `n_matrix` which it populates with the result of applying the function `mantel` to the complex numbers represented in this grid.

- In more detail, supposing that `x_matrix` is the $d_1 \times d_2$ matrix (x_{kl}) and `y_matrix` is the $d_1 \times d_2$ matrix (y_{kl}), then `n_matrix` will be the $d_1 \times d_2$ matrix (n_{kl}) where, for each k, l, component n_{kl} is the result of the computation of the function `mantel` on input $c_{kl} := x_{kl} + y_{kl}i$.

Our function then returns the 3-tuple (`x_matrix`, `y_matrix`, `n_matrix`). (The brackets are optional here.)

We have already designed most of the function. Your task is to code the action specified in the bullet point above. You should do this in the space below the line

```
# YOUR CODE HERE
```

in the cell below.

[10]:
```
def make_mandelbrot_data(xmin, xmax, num_xpts, ymin, ymax, num_ypts):
    # Global variable used (inside the call to mandel): MAXITER
    # First construct  x and y vectors, this time using the
    # function linspace.
    x_vector = np.linspace(xmin, xmax, num_xpts)
    y_vector = np.linspace(ymin, ymax, num_ypts)

    # We transform  x_vector and y_vector into matrices that together represent␣
↪a grid of coordinates
    # (See "Generating a 2 dimensional coordinate grid" in Lecture 4_2 for␣
↪explanation)
    x_matrix, y_matrix = np.meshgrid(x_vector, y_vector)

    # We create a zero matrix "n_matrix" of the same dimensions as x_matrix and␣
↪y_matrix
    rows, cols = np.shape(x_matrix)
    n_matrix = np.zeros((rows,cols), dtype=int) # Neat using dtype=int, but no␣
↪need for this

    # OVER TO YOU: YOUR CODE MUST POPULATE n_matrix WITH THE APPROPRIATE VALUES
```

```
    # AS SPECIFIED IN THE BULLET POINT ABOVE
    ### BEGIN SOLUTION
    for i in range(rows):
        for k in range(cols):
            n_matrix[i,k] = mandel(x_matrix[i,k] + y_matrix[i,k]*1j)
    # YOUR CODE SHOULD END HERE (YOU ONLY NEED 3 LINES)
    return x_matrix, y_matrix, n_matrix
    ### END SOLUTION
```

[11]:
```
# ALTERNATIVE SOLUTION - TAKING ADVANTAGE OF NUMPY FUNCTIONALITY - TO APPEAR IN␣
 ↪THE MODEL SOLUTION
# IN FACT IF WE "vectorize" THE FUNCTION mandel WE CAN GENERATE n_matrix MORE␣
 ↪DIRECTLY
# WITHOUT USE OF A "for loop" TO POPULATE IT
### BEGIN SOLUTION
# Construct a vectorized version of mandel in order to handle numpy arrays
mandel_vec = np.vectorize(mandel)

def make_mandelbrot_data(xmin, xmax, num_xpts, ymin, ymax, num_ypts):
    # Global variable used (inside the call to mandel): MAXITER
    # First construct  x and y vectors, this time using the function linspace.
    x_vector = np.linspace(xmin,xmax,num_xpts)
    y_vector = np.linspace(ymin,ymax,num_ypts)

    # We transform  x_vector and y_vector into matrices that together represent␣
 ↪a grid of coordinates
    # (See "Generating a 2 dimensional coordinate grid" in Lecture 4_2 for␣
 ↪explanation)
    x_matrix, y_matrix = np.meshgrid(x_vector,y_vector)

    n_matrix = mandel_vec(x_matrix + y_matrix * 1j)

    return x_matrix, y_matrix, n_matrix
### END SOLUTION
```

[12]:
```
# YOU CAN TEST YOUR FUNCTION HERE
NUM_POINTS = 11
make_mandelbrot_data(-1,1,NUM_POINTS,-1.,1.,NUM_POINTS)
# More of your tests here...
```

[12]:
```
(array([[-1. , -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,  0.4,  0.6,  0.8,  1. ],
        [-1. , -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,  0.4,  0.6,  0.8,  1. ],
        [-1. , -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,  0.4,  0.6,  0.8,  1. ],
        [-1. , -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,  0.4,  0.6,  0.8,  1. ],
        [-1. , -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,  0.4,  0.6,  0.8,  1. ],
        [-1. , -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,  0.4,  0.6,  0.8,  1. ],
        [-1. , -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,  0.4,  0.6,  0.8,  1. ],
```

```
       [-1. , -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,  0.4,  0.6,  0.8,  1. ],
       [-1. , -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,  0.4,  0.6,  0.8,  1. ],
       [-1. , -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,  0.4,  0.6,  0.8,  1. ],
       [-1. , -0.8, -0.6, -0.4, -0.2,  0. ,  0.2,  0.4,  0.6,  0.8,  1. ]]),
 array([[-1. , -1. , -1. , -1. , -1. , -1. , -1. , -1. , -1. , -1. , -1. ],
       [-0.8, -0.8, -0.8, -0.8, -0.8, -0.8, -0.8, -0.8, -0.8, -0.8, -0.8],
       [-0.6, -0.6, -0.6, -0.6, -0.6, -0.6, -0.6, -0.6, -0.6, -0.6, -0.6],
       [-0.4, -0.4, -0.4, -0.4, -0.4, -0.4, -0.4, -0.4, -0.4, -0.4, -0.4],
       [-0.2, -0.2, -0.2, -0.2, -0.2, -0.2, -0.2, -0.2, -0.2, -0.2, -0.2],
       [ 0. ,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ],
       [ 0.2,  0.2,  0.2,  0.2,  0.2,  0.2,  0.2,  0.2,  0.2,  0.2,  0.2],
       [ 0.4,  0.4,  0.4,  0.4,  0.4,  0.4,  0.4,  0.4,  0.4,  0.4,  0.4],
       [ 0.6,  0.6,  0.6,  0.6,  0.6,  0.6,  0.6,  0.6,  0.6,  0.6,  0.6],
       [ 0.8,  0.8,  0.8,  0.8,  0.8,  0.8,  0.8,  0.8,  0.8,  0.8,  0.8],
       [ 1. ,  1. ,  1. ,  1. ,  1. ,  1. ,  1. ,  1. ,  1. ,  1. ,  1. ]]),
 array([[  3,   3,   4,   4,   7, 100,   4,   3,   2,   2,   2],
       [  3,   4,   4,   6, 100,  18,   5,   4,   3,   2,   2],
       [  4,   5,  12,  26, 100, 100,  12,  15,   3,   3,   2],
       [  7,   7, 100, 100, 100, 100, 100,   9,   4,   3,   2],
       [100,  15, 100, 100, 100, 100, 100,  31,   4,   3,   2],
       [100, 100, 100, 100, 100, 100, 100,   7,   4,   3,   3],
       [100,  15, 100, 100, 100, 100, 100,  31,   4,   3,   2],
       [  7,   7, 100, 100, 100, 100, 100,   9,   4,   3,   2],
       [  4,   5,  12,  26, 100, 100,  12,  15,   3,   3,   2],
       [  3,   4,   4,   6, 100,  18,   5,   4,   3,   2,   2],
       [  3,   3,   4,   4,   7, 100,   4,   3,   2,   2,   2]]))
```

[13]:
```
# ANOTHER TEST ADDED FOR ILLUSTRATION. WE HAVE ZOOMED INTO A "BORDER" REGION OF␣
 ↪THE MANDELBROT SET
# REMEMBER THAT IN THE THIRD ARRAY (n_matrix) 100 (= MAXITER) MEANS THAT THE␣
 ↪COMPLEX POINT (WHOSE COORDINATES
# ARE THE CORRESPONDING COMPONENTS IN THE TWO PRECEDING ARRAYS) IS IN THE␣
 ↪MANDELBROT SET,
# ANY OTHER (LOWER) VALUE MEANS THAT IT IS NOT IN THE SET
NUM_POINTS = 5
x_matrix, y_matrix, n_matrix = make_mandelbrot_data(0.36,0.37,NUM_POINTS,0.1, 0.
 ↪11,NUM_POINTS)
print(x_matrix,"\n")
print(y_matrix,"\n")
print(n_matrix,"\n")
```

```
[[0.36   0.3625 0.365  0.3675 0.37  ]
 [0.36   0.3625 0.365  0.3675 0.37  ]
 [0.36   0.3625 0.365  0.3675 0.37  ]
 [0.36   0.3625 0.365  0.3675 0.37  ]
 [0.36   0.3625 0.365  0.3675 0.37  ]]
```

```
[[0.1    0.1    0.1    0.1    0.1   ]
 [0.1025 0.1025 0.1025 0.1025 0.1025]
 [0.105  0.105  0.105  0.105  0.105 ]
 [0.1075 0.1075 0.1075 0.1075 0.1075]
 [0.11   0.11   0.11   0.11   0.11  ]]

[[100 100 100  40 100]
 [100 100 100  63  73]
 [100 100 100 100 100]
 [100 100 100  35  33]
 [ 78  91  26  71  58]]
```

1.1.5 Part 5: plot the Mandelbrot set

You should have noticed that our function `make_mandelbrot_data` returns three numpy arrays of the type that the pyplot pixellated plotting function `pcolor` takes as its first three input arguments. The matrices/arrays returned correspond respectively to the x, y and z coordinates in 3 dimensions with the z coordinate corresponding to the output n of the function `mantel` on complex number $c = x + yi$. Remember that c is (or, more precisely, we guess that it is) in the Mandelbrot set if this output n is $=$ `MAXITER` and c is not in the set otherwise (i.e. if $n <$ `MAXITER`). The function `pcolor` calibrates the colours of each point c in the complex plane (or at least in our grid) according to the size of n and hence we can see for each point if it is either in the Mandelbrot set or otherwise "how far out" of, the Mandelbrot set it is according to the colour attributed to it.

You should now proceed by retrieving (via the function `make_mandelbrot_data` the mandelbrot data on the region $[-2, 1] \times [-1.5, 1, 5]$ of the complex plane with, for example `NUM_POINTS = 401` many coordinates for both the x and y coordinates.

You should then plot your result in a similar manner to that used in the example of Part 3, or in subsection "Pixellated plot using `pcolor`" of Lecture 4.2. We suggest that you use the colour map 'gist_ncar' (instead of the colour map 'PuOr' above or 'PiYG' used in Lecture 4.2) as this allows you to see clearly how the n values output by the `mandel` function vary. There are however lots of colour maps that you can choose from: you can copy and paste the names from the page https://matplotlib.org/stable/gallery/color/colormap_reference.html . To define the colour limits you should set `vmin = 1` and `vmax = MAXITER`. (We know that these are the minimum and maximum values returned by the function `mandel` on the region $[-2, 1] \times [-1.5, 1, 5]$ so there is no need to compute these values as we have done in other examples.)

[14]:
```
NUM_POINTS = 401 # YOU CAN TRY E.G. 501 TO GET HIGHER RESOLUTION (BUT MORE␣
 ↪SLOWLY PROCESSED)
### BEGIN SOLUTION
x_matrix, y_matrix, n_matrix = make_mandelbrot_data(-2, 1, NUM_POINTS, -1.
 ↪5, NUM_POINTS)

plt.figure(figsize=(10,8))
# We have  chosen the gist_ncar colormap. Note: vmin, vmax set the colour␣
 ↪limits.
```

```
plt.pcolor(x_matrix, y_matrix, n_matrix, cmap='gist_ncar', vmin = 1, vmax =␣
 ↪MAXITER)
# We add a colour scale bar
plt.colorbar()
plt.show()
### END SOLUTION
```