CTA200 Assignment 3

Andrew Li — 1006675625 — andrewp.li@mail.utoronto.ca

May 9, 2023

The methods used in this report will be completed through Python using Jupyter Notebook. The library `numpy` (Harris et al., 2020) is used to aid our compuation using arrays, `scipy` (Virtanen et al., 2020) is used to solve initial value problems, and `matplotlib` (Hunter, 2007) is used for plotting and visualization purposes.

1 The Mandelbrot set

In this section we explore the generation of the Mandelbrot set. Define a complex plane given by points $c = x + iy$ with $x, y \in [-2, 2]$. Set $z_0 = 0$ and iterate

$$z_{i+1} = z^2 + c \tag{1}$$

along each point in the plane. For some c, this will diverge to infinity, but others will remain bounded. The set of c that remains bounded comprise the Mandelbrot set. We will generate this set using Python.

We define a function `f(z0,c,max_iter)` that iterates equation 1. The inputs are an initial guess z_0, a complex number c, and the maximum iterations to try. This function will iterate until either the maximum iterations is reached, or the condition for divergence is met. This condition for the Mandelbrot set is when $|z_i| > 2$. If this condition is met, then the iteration is returned, otherwise, the value of c is considered to be convergent, and `np.nan` is returned. The function is stated below.

```
def f(z0,c,max_iter):
    '''Iterate z_i+1=z^2+c until divergent

    Paramters:
    z0 - A complex number. The initial guess
    c - A complex number. The value of c to use
    max_iter - An integer. The maximum iterations

    Returns:
    An integer. The number of iterations to diverge, or np.nan if convergent.
    '''
    for i in range(max_iter):
        z1 = z0**2 + c #iterate
        if np.abs(z1) > 2: # criteria for divergence, return the iteration
            return i
        else: #update
            z0 = z1
    return np.nan #converges
```

We define a function `complex_iterate(N,z0,max_iter)` that creates a uniform `N`×`N` complex plane as defined before and applies `f` to each point on this plane. The inputs are the number of grid lines, an initial guess z_0, and the maximum iterations to try. A 2d numpy array is returned. The function is stated below.

```python
def complex_iterate(N=100,z0=0,max_iter=1000):
    '''Applies f to a complex plane

    Paramters:
    N - An integer. Creates NxN grid to sample points
    z0 - A complex number. The initial guess
    max_iter - An integer. The maximum iterations

    Returns:
    2d numpy array. The coordinates of each is the number of iterations to diverge, np.nan if convergent.
    '''
    # Creating complex plane
    arr = np.linspace(-2,2,N)
    re, im = np.meshgrid(arr,arr*1j)
    c = re + im

    return [[f(z0,c[i][j],max_iter) for j in range(N)] for i in range(N)] # runs f for each point in the plane
```

We will use `N=200,max_iter=1000`. Figure 1 shows the results of this function. The leftmost plot shows the convergent values in yellow and divergent in blue. The rightmost plot shows the iteration at which the values diverged. It can be seen that the closer to the convergent values we get, the more iterations were necessary to diverge. The discreteness of the plot is imply due to `pyplot`; the values are relatively continuous.

The Mandelbrot set has been found, or at least approximated, successfully, but increasing the number of grid lines and iterations would lead to a more detailed and precise result.

2 The Lorenz equations

In Lorenz (1963), meteorologist Edward Lorenz demonstrated how a system of nonlinear differential equations can evolve into completely different behaviours due to even the slightest change in initial conditions. To demonstrate his results, he uses weather prediction as an example. Using Fourier transforms, he simplifies the governing equations into three Fourier nodes with amplitudes denoted by $W \equiv (X, Y, Z)$. The equations are given as

$$\dot{X} = -\sigma(X - Y), \tag{2}$$

$$\dot{Y} = rX - Y - XZ, \tag{3}$$

$$\dot{Z} = -bZ + XY, \tag{4}$$

where we take $\sigma = 10$, $r = 28$ and $b = 8/3$ as constants. Thus, we need to solve the equation

$$\dot{W} = \begin{pmatrix} \dot{X} \\ \dot{Y} \\ \dot{Z} \end{pmatrix} = \begin{pmatrix} -\sigma(X - Y) \\ rX - Y - XZ \\ -bZ + XY \end{pmatrix}. \tag{5}$$

To solve this equation, we will use `scipy.integrate.solve_ivp`, across 60 arbitrary time units and an initial state of `W_0=[0,1,0]`. The code used to accomplish this is shown below.

```
sigma, r, b = 10, 28, 8/3

def dW(t,W):
    '''The Lorentz equations

    Paramters:
    t - A float. Time, arbitrary in this case
    W - Array of size 3. The fourier nodes W = [X,Y,Z]

    Returns:
    An array of size 3. dW/dt
    '''
    dX = -sigma*(W[0]-W[1])
    dY = r*W[0] - W[1] - W[0]*W[2]
    dZ = -b*W[2] + W[0]*W[1]
    return [dX,dY,dZ]

W_0 = [0,1,0]
sol = solve_ivp(dW,[0,60],W_0,dense_output=True)
```

We will now replicate Figures 1 and 2 of Lorenz (1963) in Figures 2 and 3 respectively. Figure 2 shows the first 3000 iterations of Y. Although the plot differs slightly from Lorenz (1963), the overall behaviours are very similar. Before iteration 1800, the plots are nearly identical. Figure 2 shows the Z-Y plot and X-Y plot over iterations 1400-1900. Once again, the plots are nearly identical. Both plots noticeably differ by one loop on the positive Y side. This is likely because between iterations 1800 and 1900, we see positive Y in Lorenz Figure 1, whereas in Figure 2 we see Y is negative. These small differences are likely due to the different numerical methods used to solve the differential equations. Lorenz uses a double-approximation method detailed in his paper, while `solve_ivp` uses the explicit Runge-Kutta method of order 5. Since these are both approximations, the small differences between the two will become more visible as the number of iterations increases.

To demonstrate the chaotic nature of these equations, we will repeat this procedure, but instead use $W_0' = W_0 + (0, 1 \times 10^{-8}, 0)$ in place of W_0. The absolute value of the differences between the first solution and the second are shown in Figure 4. It can be seen that although the differences start small, they begin to

increase exponentially very quickly and the difference between them becomes large. The code to complete this section is shown below.

```
sol2 = solve_ivp(dW,[0,60],[0,1.00000001,0],dense_output=True)

t3 = np.arange(0,60,0.01)
W3 = sol.sol(t3).T
W3p = sol2.sol(t3).T

dist = [np.linalg.norm(i) for i in (W3-W3p)]
```

References

Harris, C. R., Millman, K. J., van der Walt, S. J., et al. 2020, Nature, 585, 357, doi: 10.1038/s41586-020-2649-2

Hunter, J. D. 2007, Computing In Science & Engineering, 9, 90, doi: 10.1109/MCSE.2007.55

Lorenz, E. N. 1963, Journal of Atmospheric Sciences, 20, 130 , doi: https://doi.org/10.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2

Virtanen, P., Gommers, R., Oliphant, T. E., et al. 2020, Nature Methods, 17, 261, doi: 10.1038/s41592-019-0686-2