Report file - Problem Set #2

Matteo Dell'Acqua

September 17, 2024

Abstract

This is the report for the problem set #2. Since the problem set is composed of five exercises, each section of this document will be divided into five subsections, one for each problem. The scripts (labelled as ps_2.'problem number') and the raw file of the images are in this directory.

1 Formulation of the problem

1.1 Problem 1

We are asked to investigate the NumPy representation of 32-bit floating point.

In NumPy, 32-bit floating point numbers are represented according to the IEEE standard. More precisely, the 32-bits are split into:

- 1 sign bit s,
- 8 bits representing the integer exponent e,
- 23 bits representing the integer mantissa f.

The actual number store is thus given by[1]:

$$(-1)^s \left(1 + \sum_{i=0}^{22} f_i 2^{i-23}\right) 2^{e-127}. \tag{1}$$

Due to the discrete nature of this representation, it is impossible tu accurately represent all the real numbers, and the stored number will often be an approximation of the original one.

1.2 Problem 2

We are asked to investigate the limits of NumPy 32- and 64-bit floating point representations.

As we previously mentioned, the floating point representation of numbers suffer from problems due to its finiteness and discreteness. To explore its limit we need to be more precise than eq. (1) and study the exception to that formula:

- $e = 0,\ f \neq 0$ are called *subnormal numbers*: the "1" above is replaced with 0, and $e - 127$ by -126;
- $e = 0,\ f = 0$ gives a signed zero;

[1]For normal numbers, see the exception in section 1.2

- $e = 255,\ f = 0$ gives a signed ∞;
- $e = 255,\ f \neq 0$ gives a `NaN` (i.e. "Not a Number").

Moreover, 64-bits floating points follows similar conventions, except that now the exponent is 11 bits long and the mantissa is 52 bits long.

1.3 Problem 3

We aim to approximate the Madelung constant of a cubic lattice:

$$M = \lim_{L\to\infty} \sum_{\substack{i,j,k=-L \\ (i,j,k)\neq(0,0,0)}}^{L} V(i,j,k) = \lim_{L\to\infty} \sum_{\substack{i,j,k=-L \\ (i,j,k)\neq(0,0,0)}}^{L} \frac{(-1)^{|i+j+k|}}{\sqrt{i^2+j^2+k^2}} \tag{2}$$

for a finite but big enough $L >> 1$. We will make use of two different methods, and compare their speed.

1.4 Problem 4

We are going to reproduce (a subregion) the Mandelbrot plot, defined as follows. Consider the function:

$$z' = z^2 + c \tag{3}$$

over the complex numbers. For a given complex number c (we are going to focus on $|c| < 2$), start with $z = 0$ and iterate the function 3. If during the iteration $|z|$ becomes greater than 2 than c is not in the Mandelbrot set, otherwise c is in the set.

1.5 Problem 5

We are asked to find the roots of a quadratic polynomial, combining to analytically equivalent expressions:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2-4ac}}{2a} \tag{4}$$

$$x_{1,2} = \frac{2c}{-b \mp \sqrt{b^2-4ac}} \tag{5}$$

and combining them to make round-off errors as small as possible.

2 Computational methods

2.1 Problem 1

Using the `get_bits` and `get_fbits` (for floats) from the Jupiter notebook of the course (which return the bit representation of numbers), we can extract the actual bits used to represent 100.98763 as a 32-foating point.

```
get_fbits(100.98763)
>>>> sign = [0]=+
     exponent = [1, 0, 0, 0, 0, 1, 0, 1]=133
     mantissa = [1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0,
                 1, 0, 1, 0, 1, 1]=4848043.
```

2.2 Problem 2

We can analytically compute the minimum increment, smallest subnormal and maximum representable number following the discussion of section 1.2. We obtain:

```
float32_resolution:
    sign = [0]=+
    exponent = [0, 1, 1, 1, 1, 1, 1, 1] = 127
    mantissa = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1] = 1

float32_smallest:
    sign = [0]=+
    exponent = [0, 0, 0, 0, 0, 0, 0, 0] = 0
    mantissa = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1] = 1

float32_max:
    sign = [0]=+
    exponent = [1, 1, 1, 1, 1, 1, 1, 0] = 254
    mantissa = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] = 8388607

float64_resolution:
    sign = [0]=+
    exponent = [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] = 1023
    mantissa = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 1] = 1

float64_smallest:
    sign = [0]=+
    exponent = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] = 0
    mantissa = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 1] = 1

float64_max:
    sign = 0
    exponent = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0] = 2046
    mantissa = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                1, 1, 1, 1]=4.5035996e+15
```

After having solved the problem analytically, we check the results by (for both `float32` and `float64`):

- comparing $1+$ `esp` with $1+z$ for $0<z<$ `esp` for the minimum increment;
- verifying that any $0<z<$ `smallest_subnormal` returns 0;
- verifying that any $z>$ `max` returns `inf`.

Alternatively, we could have used the `finfo` function of NumPy (and its attributes `esp`, `smallest_subnormal` and `max`) to obtain the same information.

2.3 Problem 3

We solve the problem both using three `for` loops (one for each variable we sum over); and exploiting the built-in attributes of the `array` type of NumPy.

Concretely, in the first case we run the loop:

```
for i in r:
    for j in r:
        for k in r:
            if i|j|k:
                M=V(i,j,k)
```

and in the second case, we first create (using the `meshgrid` function of NumPy) three 3d cubic arrays of size $(2L+1)^3$ `coordinates_x,y,z` which each stores one of the three coordinates and an empty one M whose going to store the summands of the Madelung constant. Finally, we call the `where` and `sum` attributes:

```
M=np.where(coordinates_x|coordinates_y|coordinates_z,
           -V(coordinates_x, coordinates_y, coordinates_z),0)
return(M.sum()).
```

In both cases, we implemented the $(i,j,k)\neq(0,0,0)$ constraint with the following `or`:

```
i|j|k
```

2.4 Problem 4

Due to the discrete nature of Python, we are dividing the region $|c|<2$ into a square lattice of $N=100^2$ points.

Using `meshgrid` as before we create a N^2 matrix of complex vales for c. we then proceed implementing a simultaneous `for` loop for all the values of c using the `where` attribute of arrays:

```
z=np.where(np.absolute(z)>=2, z, z**2+c).
```

Moreover, we can keep track of the time of divergence using another array as follows:

```
I=np.ones((N,N), dtype=np.float32)*(T-i)
Converge=np.where(np.absolute(z)>=2,np.maximum(I,Converge), 0.1)
```

where the maximum function assure us that `Convergence` will no longer be updated after the first time of divergence. In order to produce a more readable plot, it is better to accentuate the differences between the divergence times t_{div}. In order to do that, we chose to modify the `Convergence` array using the function:

$$(T-t_{\mathbf{div}})^3. \tag{6}$$

2.5 Problem 5

Here onward we will denote with `usual_quadratic` and `wierd_quadratic` two arrays computing the roots respectively using eq. (4) and eq. (5).

The main round-off error occurs when $|4a/b^2| << 1$ leading to $\Delta = b^2 - 4ac \sim b^2$, which in turn implies one between $-b \pm \sqrt{\Delta}$ (depending on the sign of b) is subject to subtractive cancellation error.

The interesting thing to note is that, due to the signs in eqs. (4) and (5) for each root, $x_{1,2}$ the error affects at most one of the two formulas: we can thus use the other to obtain a precise answer!

Explicitly, we can do the following:

```
if b>0:
    return np.array([usual_quadratic[0],weird_quadratic[1]])
else:
    return np.array([weird_quadratic[0],usual_quadratic[1]])
```

3 Results

3.1 Problem 1

Using eq. (1) it is now easy to reconstruct the number x which is actually stored using NumPy:

$$x = 847145664/8388608 = 100.98763275146484 \tag{7}$$

which differ from the original one by $2.7514648479609605 \cdot 10^{-6}$.

3.2 Problem 2

From the bitwise representation shown in section 2.2 we can obtain:

```
32bit resolution= 1.1920928955078125e-07
32bit smallest= 1.401298464324817e-45
32bit max= 3.4028234663852886e+38
64bit resolution= 2.220446049250313e-16
64bbit smallest= 5e-324
64bit max= 1.7976931348623157e+308
```

3.3 Problem 3

For $L = 100$ both methods output extremely similar results (as expected):

$$M_{\text{for}} = 1.7418198158396654 \qquad M_{\text{array}} = 1.7418198158362388 \tag{8}$$

but they take very different time to run:

$$t_{\text{for}} = 69.81823770003393s \qquad t_{\text{array}} = 0.5477509999182075s. \tag{9}$$

The difference (of two orders of magnitude) is due to the fact that while in the `for` loop, the computer needs to verify every time whether the two quantities we are adding are of the same type, a NumPy `array` is guaranteed to store values of only one type.

3.4 Problem 4

In the following, we report the standard Mandelbrot plot for $T = 100$, as well as the one which keeps track of the time of divergence. They are created using the `pcolormesh` attribute of `matplotlib.pyplot`.

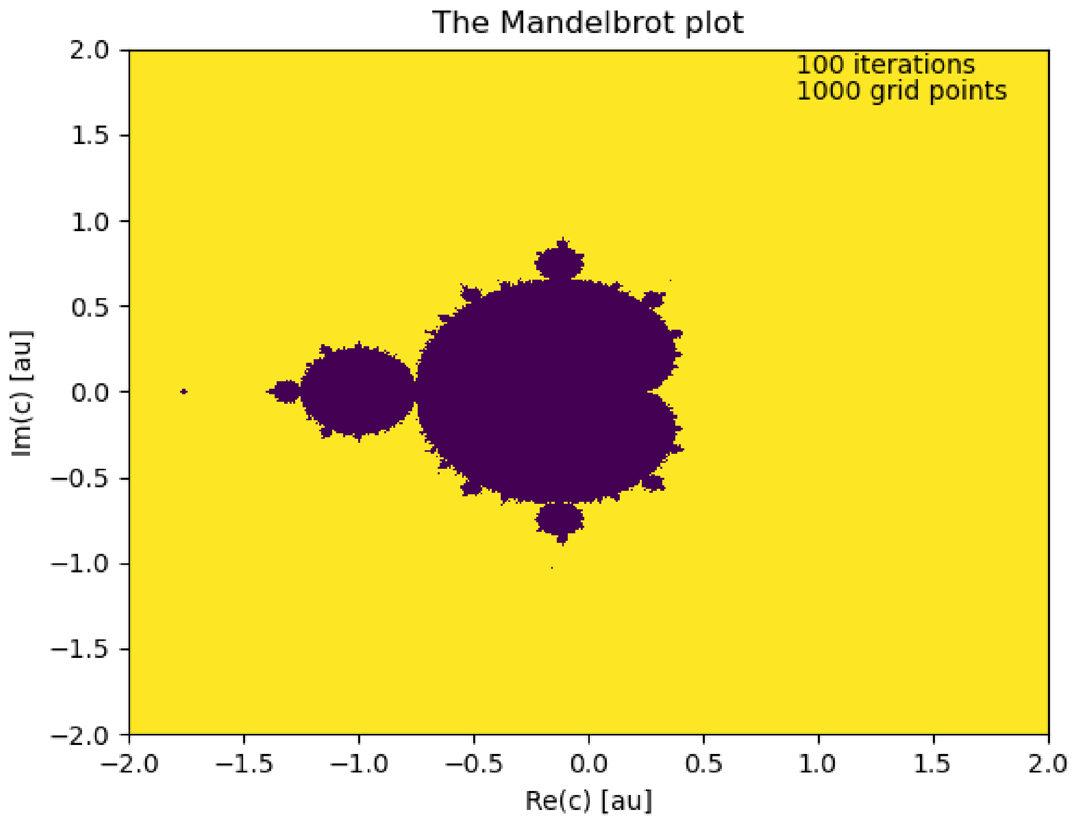

Figure 1: The Mandelbrot plot in the $|c| < 2$ region.

3.5 Problem 5

We manage to run the test designed, confirming that our function can compute roots with a precision of more than 10^{-10}

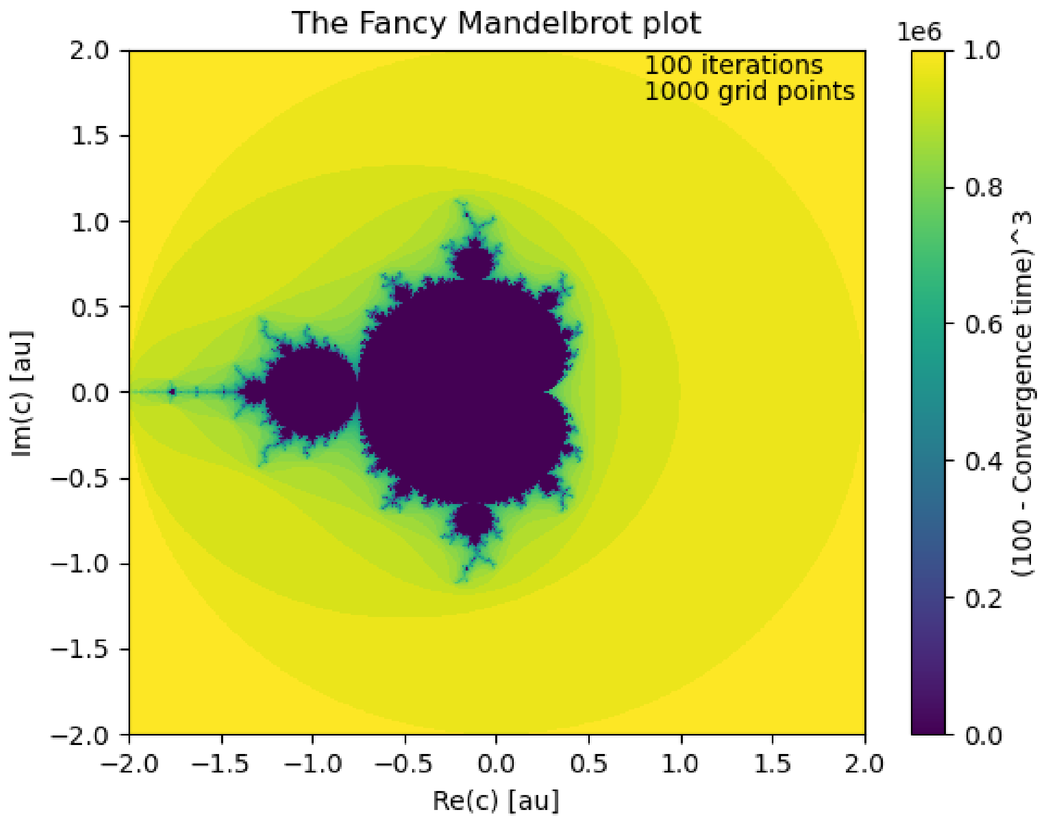

Figure 2: The modified Mandelbrot plot in the $|c| < 2$ region. We colour each value of c depending on its divergence time $t_{\mathbf{div}}$ (using the function $(T - t_{\mathbf{div}})^3$