Coding Example: The Mandelbrot Set (NumPy approach)

In this lesson, we are going to look at two NumPy approaches to solve this case study!

We'll cover the following

Solution 1: NumPy Implementation

The trick is to search at each iteration values that have not yet diverged and update relevant information for these values and only these values. Because we start from Z = 0, we know that each value will be updated at least once (when they're equal to 0, they have not yet diverged) and will stop being updated as soon as they've diverged. To do that, we'll use NumPy fancy indexing with the less(x1,x2) function that return the truth value of (x1 < x2) element-wise.

```
def mandelbrot_numpy(xmin, xmax, ymin, ymax, xn, yn, maxiter, horizon=2.0):
  X = np.linspace(xmin, xmax, xn, dtype=np.float32)
  Y = np.linspace(ymin, ymax, yn, dtype=np.float32)
  C = X + Y[:,None]*1j
  N = np.zeros(C.shape, dtype=int)
  Z = np.zeros(C.shape, np.complex64)
  for n in range(maxiter):
    I = np.less(abs(Z), horizon)
    N[I] = n
    Z[I] = Z[I]**2 + C[I]
  N[N == maxiter-1] = 0
  return Z, N
```

Now lets replace the python approach with this one and see what happens:

main.py
tools.py

```
# -----------------------------------------------------------------------------
# From Numpy to Python
# Copyright (2017) Nicolas P. Rougier - BSD license
# More information at https://github.com/rougier/numpy-book
# -----------------------------------------------------------------------------

import numpy as np

def mandelbrot_numpy1(xmin, xmax, ymin, ymax, xn, yn, maxiter, horizon=2.0):
    # Adapted from https://www.ibm.com/developerworks/community/blogs/jfp/...
    #              .../entry/How_To_Compute_Mandelbrodt_Set_Quickly?lang=en
    X = np.linspace(xmin, xmax, xn, dtype=np.float32)
    Y = np.linspace(ymin, ymax, yn, dtype=np.float32)
    C = X + Y[:,None]*1j
    N = np.zeros(C.shape, dtype=int)
    Z = np.zeros(C.shape, np.complex64)
    for n in range(maxiter):
        I = np.less(abs(Z), horizon)
        N[I] = n
        Z[I] = Z[I]**2 + C[I]
    N[N == maxiter-1] = 0
    return Z, N

def mandelbrot(xmin, xmax, ymin, ymax, xn, yn, itermax, horizon=2.0):
    # Adapted from
    # https://thesamovar.wordpress.com/2009/03/22/fast-fractals-with-python-and
    Xi, Yi = np.mgrid[0:xn, 0:yn]
    Xi, Yi = Xi.astype(np.uint32), Yi.astype(np.uint32)
```

Here is the benchmark:

```
timeit("mandelbrot_python(xmin, xmax, ymin, y max, xn, yn, maxiter)", globals())
#1 loops, best of 3: 6.1 sec per loop
timeit("mandelbrot_numpy1(xmin, xmax, ymin, ymax, xn, yn, maxiter)", globals())
# 1 loops, best of 3: 1.15 sec per loop
```

Here we can see that the time taken by mandelbrot_numpy1 is less as compared to time taken by mandelbrot_python.

Solution 2: NumPy Implementation (Faster)

The gain is roughly a 5x factor, not as much as we could have expected. Part of the problem is that the `np.less` function implies $xn \times yn$ tests at every iteration while we know that some values have already diverged. Even if these tests are performed at the C level (through NumPy), the cost is nonetheless significant.

Another approach proposed by Dan Goodman is to work on a dynamic array at each iteration that stores only the points which have not yet diverged. It requires more lines but the result is faster and leads to a 10x factor speed improvement compared to the Python version.

```
def mandelbrot_numpy_2(xmin, xmax, ymin, ymax, xn, yn, itermax, horizon=2.0):
  Xi, Yi = np.mgrid[0:xn, 0:yn]
  Xi, Yi = Xi.astype(np.uint32), Yi.astype(np.uint32)
  X = np.linspace(xmin, xmax, xn, dtype=np.float32)[Xi]
  Y = np.linspace(ymin, ymax, yn, dtype=np.float32)[Yi]
  C = X + Y*1j
  N_ = np.zeros(C.shape, dtype=np.uint32)
  Z_ = np.zeros(C.shape, dtype=np.complex64)
  Xi.shape = Yi.shape = C.shape = xn*yn

  Z = np.zeros(C.shape, np.complex64)
  for i in range(itermax):
    if not len(Z): break

    # Compute for relevant points only
    np.multiply(Z, Z, Z)
    np.add(Z, C, Z)

    # Failed convergence
    I = abs(Z) > horizon
    N_[Xi[I], Yi[I]] = i+1
    Z_[Xi[I], Yi[I]] = Z[I]

    # Keep going with those who have not diverged yet
    np.negative(I,I)
    Z = Z[I]
    Xi, Yi = Xi[I], Yi[I]
    C = C[I]
```

The difference between both approaches(mandelbrot_numpy1 and mandelbrot_numpy2) is highlighted in the code below. Replacing the previous code with this faster approach:

main.py
tools.py

```
# -----------------------------------------------------------------------------
# From Numpy to Python
# Copyright (2017) Nicolas P. Rougier - BSD license
# More information at https://github.com/rougier/numpy-book
# -----------------------------------------------------------------------------

import numpy as np

def mandelbrot_numpy2(xmin, xmax, ymin, ymax, xn, yn, itermax, horizon=2.0):
    # Adapted from
    # https://thesamovar.wordpress.com/2009/03/22/fast-fractals-with-python-and
    Xi, Yi = np.mgrid[0:xn, 0:yn]
    Xi, Yi = Xi.astype(np.uint32), Yi.astype(np.uint32)
    X = np.linspace(xmin, xmax, xn, dtype=np.float32)[Xi]
    Y = np.linspace(ymin, ymax, yn, dtype=np.float32)[Yi]
    C = X + Y*1j
    N_ = np.zeros(C.shape, dtype=np.uint32)
    Z_ = np.zeros(C.shape, dtype=np.complex64)
    Xi.shape = Yi.shape = C.shape = xn*yn

    Z = np.zeros(C.shape, np.complex64)
    for i in range(itermax):
        if not len(Z):
            break

        # Compute for relevant points only
        np.multiply(Z, Z, Z)
        np.add(Z, C, Z)
```

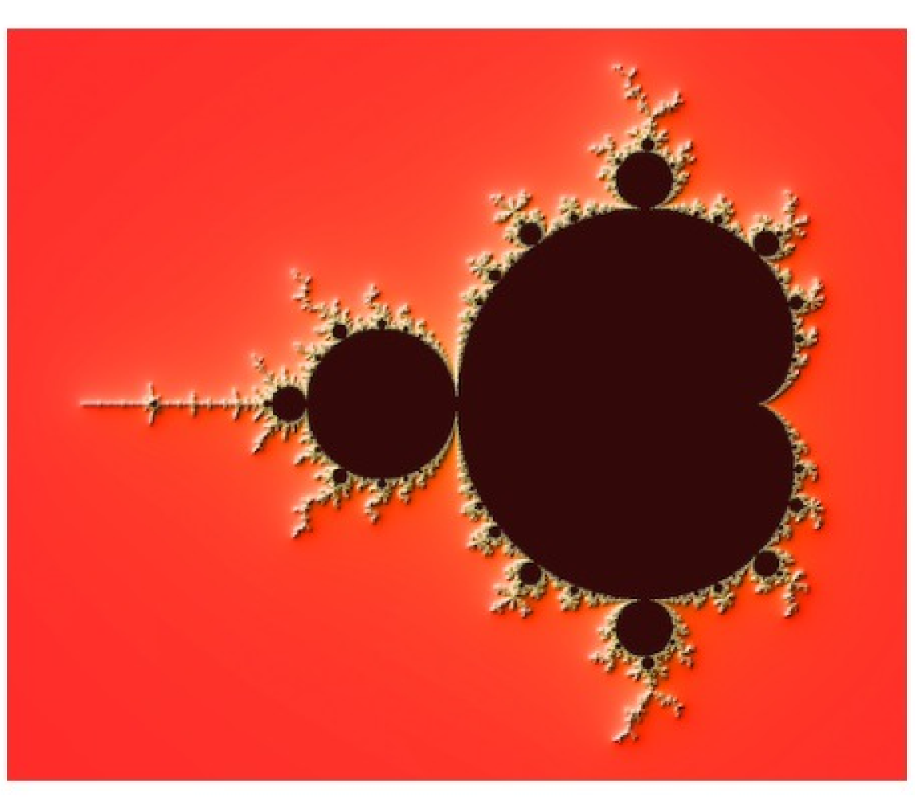
The benchmark gives us:

```
timeit("mandelbrot_numpy2(xmin, xmax, ymin, ymax, xn, yn, maxiter)", globals())
# 1 loops, best of 3: 510 msec per loop
```

Here we can see that by using the mandelbrot_numpy2 function the time is even less as compared to mandelbrot_numpy1.

Visualization

In order to visualize our results, we could directly display the N array using the `matplotlib imshow` command, but this would result in a "banded" image that is a known consequence of the escape count algorithm that we've been using. Such banding can be eliminated by using a fractional escape count. This can be done by measuring how far the iterated point landed outside of the escape cutoff. See the reference below about the renormalization of the escape count. Here is a picture of the result where we use recount normalization, and add a power normalized color map (gamma=0.3) as well as light shading.



Solve this Quiz!

Q What does `numpy.less()` do?

A) Return the truth value of (x1 < x2) element-wise.

B) Return the truth value of (x1 > x2) element-wise.

COMPLETED 0%   1 of 1

Now, let's move on to another coding example in the next lesson.