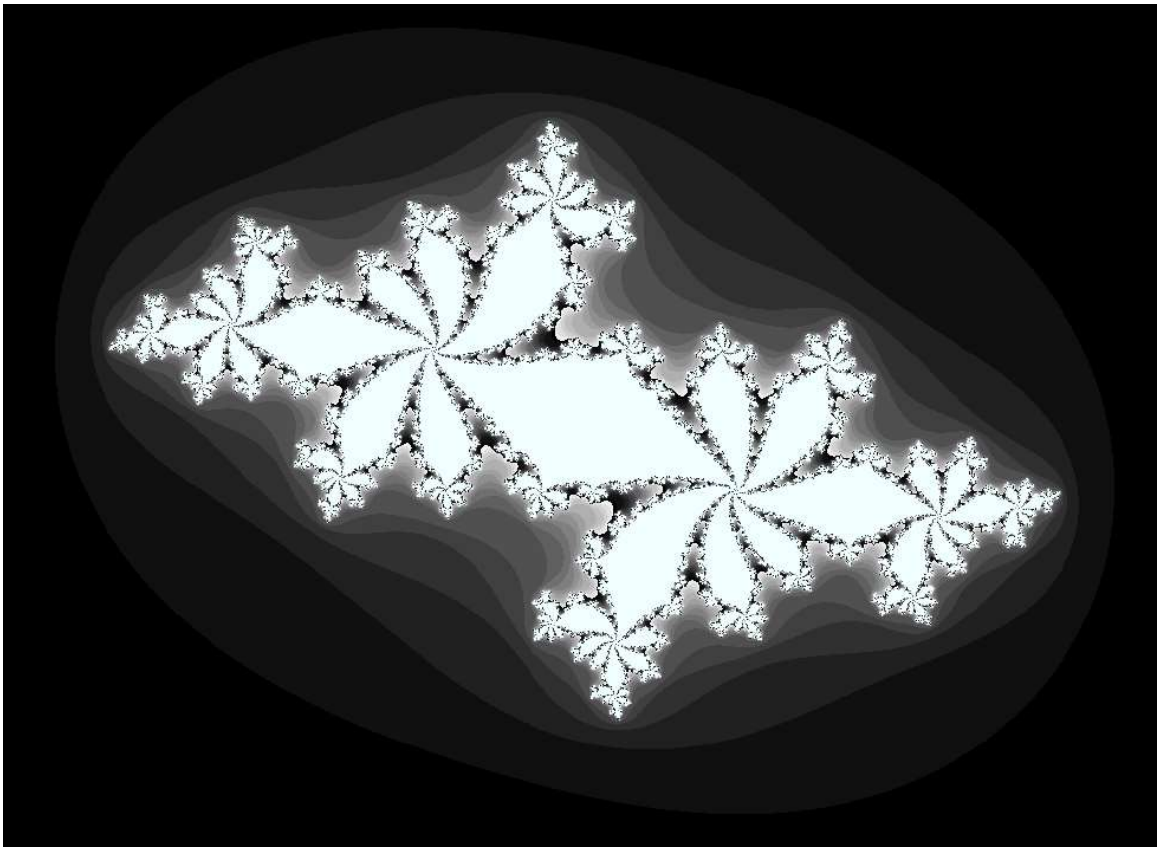
Julia Set

Description

This example has been taken from the book "*High Performance Python: Practical Performant Programming for Humans*" by Micha Gorelick and Ian Ozsvald, O'Reilly Ed., 2014.

The Julia set is a fractal sequence that generates a complex output image, named after Gaston Julia. The next figure presents a Julia set plot with a false grayscale to highlight detail. It is an interesting CPU-bound problem with a very explicit set of inputs, which allows us to profile both the CPU usage and the RAM usage so we can understand which parts of our code are consuming two of our scarce computing resources. This implementation is *deliberately* suboptimal, so we can identify memory-consuming operations and slow statements. https://en.wikipedia.org/wiki/Julia_set



The problem is interesting because we calculate each pixel by applying a loop that could be applied an **indeterminate number of times**. On each iteration, we **test** to see if this coordinate's value escapes toward infinity, or if it seems to be held by an attractor. Coordinates that cause few iterations are colored darkly in the figure, and those that cause a high number of iterations are colored white. White regions are more complex to calculate and so take longer to generate.

We define a set of *z*-coordinates that we will test. The function that we calculate squares the complex number **z** and adds **c**:

$$f(z) = z^2 + c$$

We iterate on this function while testing to see if the escape condition holds using the **abs**() function. If the escape function is False, then we break out of the loop and record the number of iterations we performed at this coordinate. If the escape function is never False, then we stop after **maxiter** iterations. We will later turn this **z**'s result into a colored pixel representing this complex location.

In pseudocode, it might look like:

```
for z in coordinates:
  for iteration in range(maxiter): # limited iterations per point
    if abs(z) < 2.0:               # has the escape condition been broken?
      z = z*z + c
    else:
      break
    # store the iteration count for each z and draw later
```

Baseline Python program

We propose the following function to implement the Julia update rule for a certain lists of complex numbers, `zs` and `cs`, with a maximum number of iterations defined as `maxiter`. The lists are the same size, and the size of the lists is the total number of pixels in the output

image. We will make executions with images of 1000 x 1000 pixels, which means that the Python sentences inside the **for** loop are executed a million times (and the sentences inside the **while** loop can then be executed much more times).

```
def calculate_z (maxiter, zs, cs):
  """Calculate output list using Julia update rule"""
  output = [0] * len(zs)
  for i in range(len(zs)):
    n = 0
    z = zs[i]
    c = cs[i]
    while abs(z) < 2 and n < maxiter:
      z = z * z + c
      n += 1
    output[i] = n
  return output
```

We use **ipython** (remember to configure de Miniconda version of Python in the LAB computers) with the magic function **%timeit** (options: -r 3 –n 1) to measure the specific execution time of the function calc_pure_python in the **aolin-login** processor. The difference of the execution time of the specific function with respect to the user execution time of the whole program is around 1%.

```
In [1]: import Julia as J
In [2]: %timeit -r 3 -n 1 J.calc_pure_python(1000,300) # -n 3 runs, -r 1 loop each
Length of x: 1000
Total elements: 1000000
Length of x: 1000
Total elements: 1000000
Length of x: 1000
Total elements: 1000000
8 s ± 84.1 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)
```

We can use the **cProfile** utility by using **ipython** with the magic function **%prun** to find out where the execution time is mostly spent during the execution of the program. Notice that the total execution time increases from 8 to 11.8 seconds: the 3.8-second penalty is due to the overhead of the code required to instrument the execution for profiling.

```
In [3]: %prun -s cumtime J.calc_pure_python(1000,300)
Length of x: 1000
Total elements: 1000000
         36221992 function calls in 11.819 seconds
   Ordered by: cumulative time
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   11.834   11.834 {built-in method builtins.exec}
        1    0.034    0.034   11.834   11.834 Julia.py:2(<module>)
        1    0.615    0.615   11.801   11.801 Julia.py:5(calc_pure_python)
        1    8.125    8.125   11.057   11.057 Julia.py:43(calculate_z)
 34219980    2.932    0.000    2.932    0.000 {built-in method builtins.abs}
  2002000    0.122    0.000    0.122    0.000 {method 'append' of 'list' objects}
        1    0.007    0.007    0.007    0.007 {built-in method builtins.sum}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        4    0.000    0.000    0.000    0.000 {built-in method builtins.len}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

More than 36 million function calls occur, most of them to the funtion **builtin.abs**: we could not predict in advance exactly how many calls would be made to **abs**, since the Julia function has unpredictable dynamics (that is why it is so interesting to look at). At best, we could have said that it will be called a minimum of 1 million times (one per pixel), and at most 300 million times (the maximum number of iterations per pixel). Therefore, 34 million calls is roughly 10% of the worst case (and white regions in the image will account for roughly 10% of the image). Around 615 miliseconds are spent on lines of code inside function calc_pure_python, not including the time spent by the CPU-intensive calculate_z function. 122 miliseconds of this time are required to call the method '**append**' of a list.

Challenge for the LAB class

Optimize the code to generate the **fastest** execution that you can devise. The idea is to "*cythonize*" the python code, so that you can use the advantage of most of the automatic optimization strategies provided by the gnu C compiler (**gcc**). Remember to use the best compiler available and to provide useful information in the program script to help the compilers (Cython and C) do their task, both via **type declarations** and **cython directives** to disable some runtime checks,. Some transformations on the type and sequence of operations in the original code and on the control sentences can also be useful. Those transformations must maintain the same **program's functionality**: i.e., the **program result must** be the same for all the possible inputs.