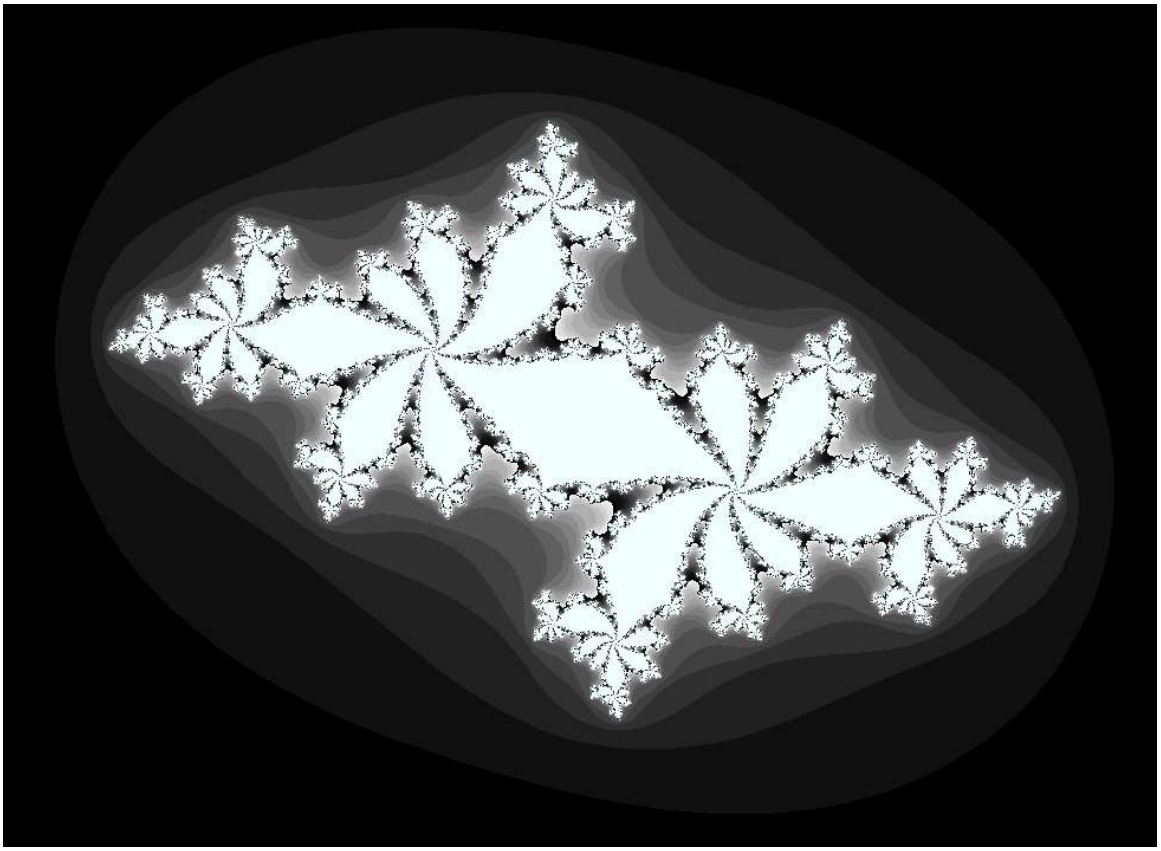
Julia Set

Universitat Autònoma de Barcelona

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. Later we will fix a slow logic statement and a memory-consuming statement. 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
```

Julia Set

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). The whole program can be obtained and inspected on the Moodle Lesson of the course.

```
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 can use the system's **time** command to get many performance system's data from the program's execution. Beware of indicating the full path (**/usr/bin/time**) instead of the simpler (and less useful) version built into our shell. The most useful information is the CPU usage, the major/minor page faults, the resident set size, number of context switches, and the file input/output.

```
[jcmoure@aolin-login Julia]$ /usr/bin/time --verbose python Julia.py # include route to system's time
Length of x: 1000
Total elements: 1000000
        Command being timed: "python Julia.py"
        User time (seconds): 8.10
        System time (seconds): 0.03
        Percent of CPU this job got: 99%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:08.15
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 95768
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0              # very costly: involves reading from disk
        Minor (reclaiming a frame) page faults: 24963     # costly: involves reading from main memory
        Voluntary context switches: 10
        Involuntary context switches: 21
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

We use **ipython** with the magic function **%timeit** (-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 1,23% (computed as (8,1 – 8) / 8,1 = 0,0123).

```
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 to find out where the execution time is mostly spent during the execution of the program. We can do it from the Linux Shell as indicated next. However, we can also obtain almost the same results by using **ipython** with the magic function **%prun**. Notice the usage of the **-s cumtime** option, which tells the profiler to sort the output information by cumulative time spent inside each function. Also, notice that the total execution time increases from 8.1 to 11.8 seconds: the 3.7-second penalty is due to the overhead of the code required to instrument the execution for profiling and measuring how long each function takes to execute.

Julia Set


```
[jcmoure@aolin-login Julia]$ python -m cProfile -s cumtime Julia.py # sort output by cumulative time
Length of x: 1000
Total elements: 1000000
         36221992 function calls in 11.834 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}
```

```
In [1]: import Julia as J
In [2]: %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  ...
```

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.

We can obtain much more detail by using the line profiler. The simplest way is probably using the **ipython** interpreter as indicated bellow. Notice that now the instrumentation of the code adds a substantial amount to the runtime (from 11.8 seconds using cProfile to 54 seconds using the line profiling option).

```
In [3]: %load_ext line_profiler
In [4]: %lprun -f J.calculate_z J.calc_pure_python(1000,300)
Length of x: 1000
Total elements: 1000000
Timer unit: 1e-06 s

Total time: 53.9916 s
File: /home/caos/jcmoure/Algorithms/Julia/Julia.py
Function: calculate_z at line 43

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    43                                           def calculate_z (maxiter, zs, cs):
    44                                            """Calculate output list using Julia update rule"""
    45         1       2180.0   2180.0      0.0    output = [0] * len(zs)
    46   1000001     405873.0      0.4      0.8    for i in range(len(zs)):
    47   1000000     388720.0      0.4      0.7      n = 0
    48   1000000     444997.0      0.4      0.8      z = zs[i]
    49   1000000     415572.0      0.4      0.8      c = cs[i]
    50  34219980   20678810.0      0.6     38.3      while abs(z) < 2 and n < maxiter:
    51  33219980   16819828.0      0.5     31.2        z = z * z + c
    52  33219980   14376340.0      0.4     26.6        n += 1
    53   1000000     459241.0      0.5      0.9      output[i] = n
    54         1          1.0      1.0      0.0    return output
```

The **%Time** column is the most useful: 38.3% of the time is spent on the **while** testing. Inside the **while** loop, we see that the updates to variables **z** and **n** are also expensive. Python's *dynamic lookup* machinery is at work for every loop, even though the same types are used for each variable in each loop iteration. This is where *type specialization* and *compilation* can gives us a massive win. The creation of the output list and the updates, respectively in lines 45 and 53, are relatively cheap compared to the cost of the while loop.