High Performance Python

Juliana Ramayo
Data Engineering
Universidad Politécnica de Yucatán
Ucú, Yucatán, México
2109128@upy.edu.mx

I. Benchmarking and Profiling

The Julia set is an interesting CPU-bound problem, it is a fractal sequence that generates a complex output image, named after Gaston Julia. After reading the sections ”Introducing the Julia Set” and ”Calculating the Full Julia Set” on Chapter 2 from the book High Performance Python [1], I implemented the chapter functions in order to calculate the Julia Set and make a representation for the false gray and pure gray scale in a Jupyter Notebook and a .py file.

To generate the plot, two lists of input data are created. The first is ‘zs‘ (complex z-coordinates) and the second is ‘cs‘ (a complex initial condition). To build them, it is necessary to know the coordinates for each z. In the code, those coordinates are build using ‘xcoord‘ and ‘ycoord‘ and a specified ‘x_step‘. Next, the ‘calculate_z_serial_purepython‘ function is defined. Also, an ‘output‘ list is defined at the start that has the same length as the input ‘zs‘ and ‘cs‘ lists. A calculation routine is called. By wrapping it in a ‘__main__‘ check, the module can be safely imported without starting the calculations for some of the profiling methods.

Then, I added the representation for false gray and pure grayscale, to achive it I used the ‘PIL‘ module because of its ability to support graphical output and the ‘array‘ module, used for its efficiency in handling large sequences of numbers. The ‘false_grayscale‘ fuction creates a false grayscale image representation of the Julia set. The false grayscale refers to a grayscale image where the colors are derived from the number of iterations needed to determine if a point escapes to infinity according to the Julia set rules. In a true grayscale image, each pixel’s color is a shade of gray, varying from black at the weakest intensity to white at the strongest intensity. However, in false grayscale, the intensity is often translated into a colorized scale (not necessarily shades of gray) to highlight differences between areas with different escape times. This function works by calculating a visualization of the Julia set with a specific width and maximum number of iterations. It starts by generating the Julia set using the calc_pure_python function, then normalizes the iteration counts to fit within a grayscale range of 0-255. This normalization allows for the mapping of iteration counts to grayscale intensity values, with higher iterations corresponding to lighter shades. Next, the function converts these grayscale values into RGB values, which are identical across all three color channels because it’s a grayscale image. The conversion is done in such a way that each grayscale value is spread across the red, green, and blue components to create a 32-bit integer representing an RGB color. After converting these RGB values into an array of unsigned integers, the function creates a new image with the given width and fills it with the RGB values to produce the visual representation of the Julia set. Finally, it displays the image.

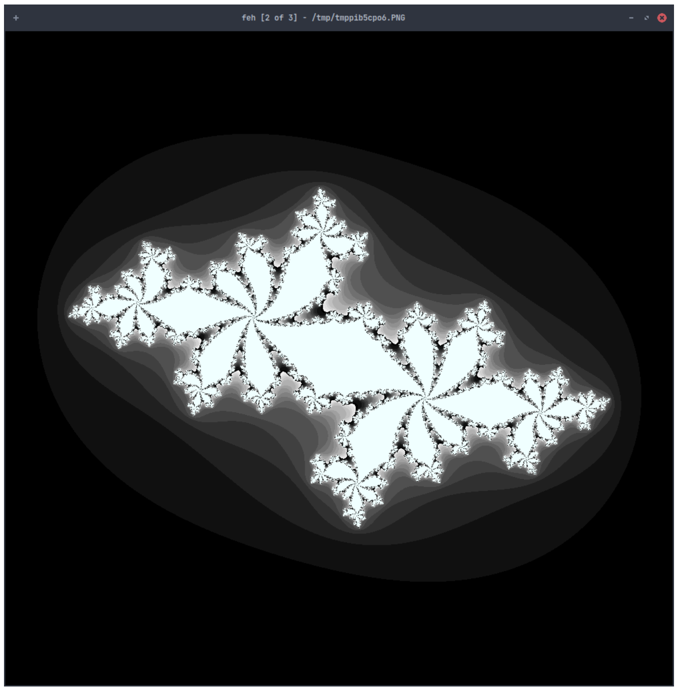


Fig. 1. Example of the false grayscale representation

The ‘pure_grayscale‘ function creates a pure grayscale image, where every pixel is represented by a single intensity value from 0 (black) to 255 (white). The number of iterations required for each point in the Julia set calculation directly maps to a grayscale intensity. Points that take more iterations to reach the escape condition are assigned a higher value (lighter), and those that escape quickly are assigned a lower value (darker). This function begins with generating the Julia set using calc_pure_python too. It then normalizes the iteration counts to grayscale values between 0-255, similar to false_grayscale, using the maximum iteration count as the normalization factor. However, unlike false_grayscale, which constructs RGB values by replicating the grayscale value across three color channels, pure_grayscale creates an array of unsigned bytes where each byte directly represents the intensity of a single pixel in grayscale. The resulting image is thus a pure grayscale, where each pixel's color is a single byte denoting shades of gray without separate color channels.

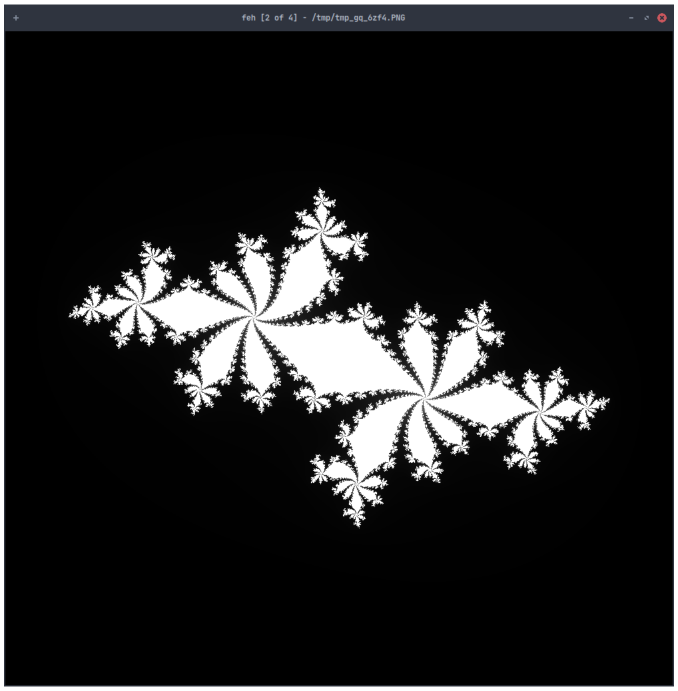
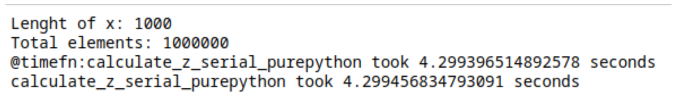


Fig. 2. Example of the pure grayscale representation

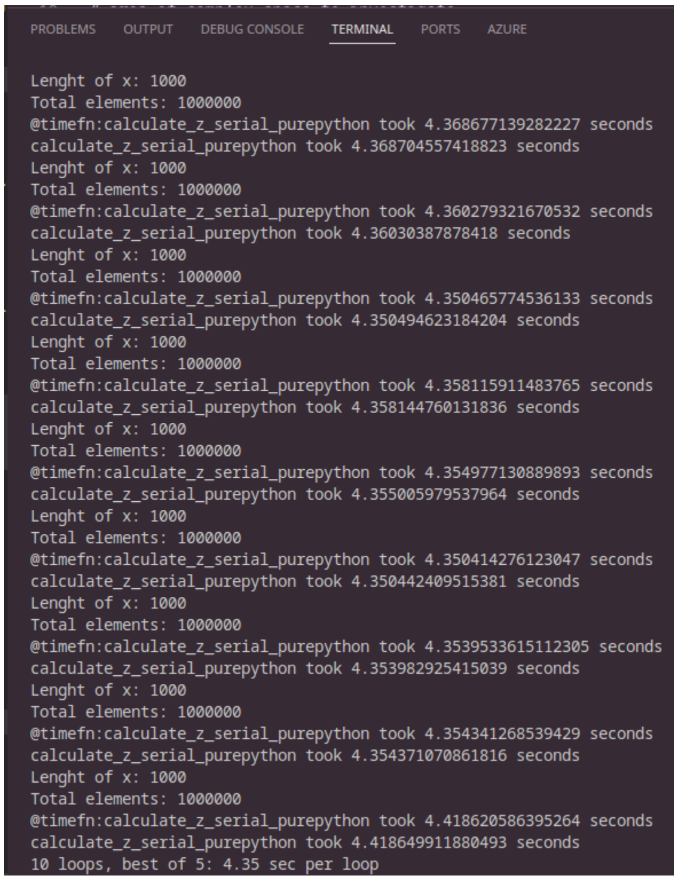
After this, I started profiling the function to understand its functionality. Firstly, I defined a new function 'timefn' which takes a function as an argument: the inner function, 'measure_time', takes '*args' (a variable number of positional arguments) and '*kwargs' (a variable number of key/value arguments) and passes them through to 'fn' for exection. Then, I decorated the 'calculate_z_serial_purepython' with '@timefn'to profile it and I obtained the following result after one test:



```
Lenght of x: 1000
Total elements: 1000000
@timefn:calculate_z_serial_purepython took 4.299396514892578 seconds
calculate_z_serial_purepython took 4.299456834793091 seconds
```

Fig. 3. Result of timefn function used in the 'calculate_z_serial_purepython'

Then, I used the 'timeit' module to get a coarse measurement of the execution speed of the CPU-bound function. Running 10 loops with 5 repetitions. Executing the measurement on the command line with 'python -m timeit -r 5 -n 10 -s "from benchmarking_and_profiling import calc_pure_python" "calc_pure_python(desired_width=1000, max_iterations=300)"' gave the following results:

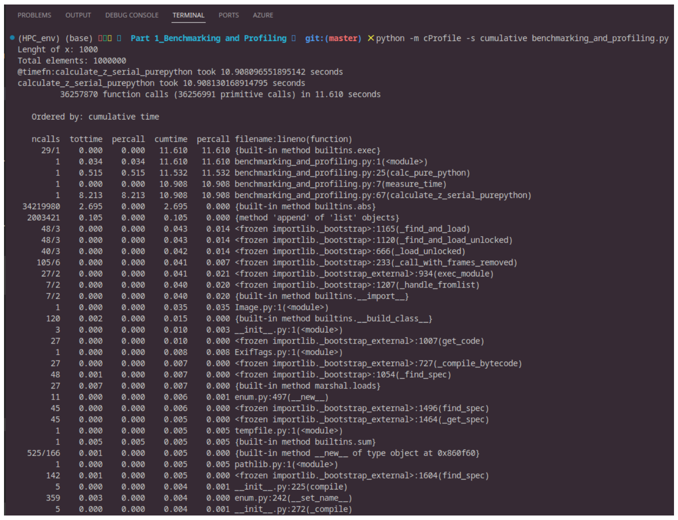


```
Lenght of x: 1000
Total elements: 1000000
@timefn:calculate_z_serial_purepython took 4.368677139282227 seconds
calculate_z_serial_purepython took 4.368704557418823 seconds
Lenght of x: 1000
Total elements: 1000000
@timefn:calculate_z_serial_purepython took 4.360279321670532 seconds
calculate_z_serial_purepython took 4.36030387878418 seconds
Lenght of x: 1000
Total elements: 1000000
@timefn:calculate_z_serial_purepython took 4.350465774536133 seconds
calculate_z_serial_purepython took 4.350494623184204 seconds
Lenght of x: 1000
Total elements: 1000000
@timefn:calculate_z_serial_purepython took 4.358115911483765 seconds
calculate_z_serial_purepython took 4.358144760131836 seconds
Lenght of x: 1000
Total elements: 1000000
@timefn:calculate_z_serial_purepython took 4.354977130889893 seconds
calculate_z_serial_purepython took 4.355005979537964 seconds
Lenght of x: 1000
Total elements: 1000000
@timefn:calculate_z_serial_purepython took 4.350414276123047 seconds
calculate_z_serial_purepython took 4.350442409515381 seconds
Lenght of x: 1000
Total elements: 1000000
@timefn:calculate_z_serial_purepython took 4.3539533615112305 seconds
calculate_z_serial_purepython took 4.353982925415039 seconds
Lenght of x: 1000
Total elements: 1000000
@timefn:calculate_z_serial_purepython took 4.354341268539429 seconds
calculate_z_serial_purepython took 4.354371070861816 seconds
Lenght of x: 1000
Total elements: 1000000
@timefn:calculate_z_serial_purepython took 4.418620586395264 seconds
calculate_z_serial_purepython took 4.418649911880493 seconds
10 loops, best of 5: 4.35 sec per loop
```

Fig. 4. Result of timeit with calc_pure_python(desired_width=1000, max_iterations=300)

To profile the code I used the 'cProfile'. I sorted the results by the time spent inside each function to obtain a view into the slowest parts. To get the results using the command line I used:

'$python -m cProfile -s cumulative benchmarking_and_profiling.py'



```
(HPC_env) (base) Part 1_Benchmarking and Profiling git:(master) ✗python -m cProfile -s cumulative benchmarking_and_profiling.py
Lenght of x: 1000
Total elements: 1000000
@timefn:calculate_z_serial_purepython took 10.908096551895142 seconds
calculate_z_serial_purepython took 10.908130168914795 seconds
         36257870 function calls (36256991 primitive calls) in 11.610 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     29/1    0.000    0.000   11.610   11.610 {built-in method builtins.exec}
        1    0.034    0.034   11.610   11.610 benchmarking_and_profiling.py:1(<module>)
        1    0.515    0.515   11.532   11.532 benchmarking_and_profiling.py:25(calc_pure_python)
        1    0.000    0.000   10.908   10.908 benchmarking_and_profiling.py:7(measure_time)
        1    8.213    8.213   10.908   10.908 benchmarking_and_profiling.py:67(calculate_z_serial_purepython)
 34219980    2.695    0.000    2.695    0.000 {built-in method builtins.abs}
  2003421    0.105    0.000    0.105    0.000 {method 'append' of 'list' objects}
     48/3    0.000    0.000    0.043    0.014 <frozen importlib._bootstrap>:1165(_find_and_load)
     48/3    0.000    0.000    0.043    0.014 <frozen importlib._bootstrap>:1120(_find_and_load_unlocked)
     40/3    0.000    0.000    0.042    0.014 <frozen importlib._bootstrap>:666(_load_unlocked)
    105/6    0.000    0.000    0.041    0.007 <frozen importlib._bootstrap>:233(_call_with_frames_removed)
     27/2    0.000    0.000    0.041    0.021 <frozen importlib._bootstrap_external>:934(exec_module)
      7/2    0.000    0.000    0.040    0.020 <frozen importlib._bootstrap>:1207(_handle_fromlist)
      7/2    0.000    0.000    0.040    0.020 {built-in method builtins.__import__}
        1    0.000    0.000    0.035    0.035 Image.py:1(<module>)
      120    0.002    0.000    0.015    0.000 {built-in method builtins.__build_class__}
        3    0.000    0.000    0.010    0.003 __init__.py:1(<module>)
       27    0.000    0.000    0.010    0.000 <frozen importlib._bootstrap_external>:1007(get_code)
        1    0.000    0.000    0.008    0.008 ExifTags.py:1(<module>)
       27    0.000    0.000    0.007    0.000 <frozen importlib._bootstrap_external>:727(_compile_bytecode)
       48    0.001    0.000    0.007    0.000 <frozen importlib._bootstrap>:1054(_find_spec)
       27    0.007    0.000    0.007    0.000 {built-in method marshal.loads}
       11    0.000    0.000    0.006    0.001 enum.py:497(__new__)
       45    0.000    0.000    0.006    0.000 <frozen importlib._bootstrap_external>:1496(find_spec)
       45    0.000    0.000    0.005    0.000 <frozen importlib._bootstrap_external>:1464(_get_spec)
        1    0.000    0.000    0.005    0.005 tempfile.py:1(<module>)
        1    0.005    0.005    0.005    0.005 {built-in method builtins.sum}
  525/166    0.001    0.000    0.005    0.000 {built-in method __new__ of type object at 0x860f60}
        1    0.000    0.000    0.005    0.005 pathlib.py:1(<module>)
      142    0.001    0.000    0.005    0.000 <frozen importlib._bootstrap_external>:1604(find_spec)
        5    0.000    0.000    0.004    0.001 __init__.py:225(compile)
      359    0.003    0.000    0.004    0.000 enum.py:242(__set_name__)
        5    0.000    0.000    0.004    0.001 __init__.py:272(_compile)
```

Fig. 5. Result of cProfile module

The profiling indicates a total of 36257870 calls in 11.61 seconds. By the image, we can see that the entry point to the code takes a total of 11 seconds. Inside ‘calc_pure_python‘, the call ‘calculate_z_serial_purepython‘ consumes 10.90 seconds, and both functions are only called once. Inside ‘calculate_z_serial_purepython‘, the time spent on lines of code is 8.21 seconds. This function makes 34219980 calls to abs, which take a total of 2.69 seconds, along with some other calls that do not cost much time.

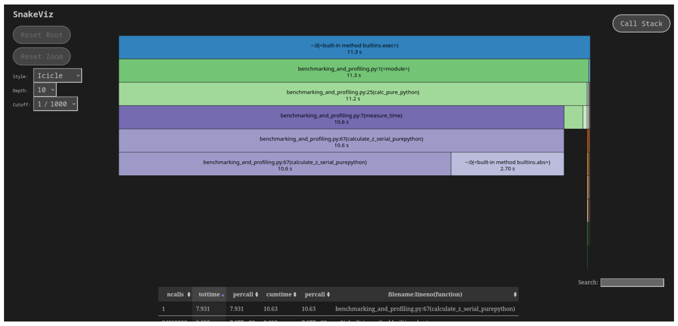
Then, I used ‘snakeviz‘ to get a high-level understanding of the cProfile statistics file.

To run ‘snakeviz‘ correctly I used the following commands:

‘$ python -m cProfile -o benchmarking_and_profiling.cprof benchmarking_and_profiling.py‘

‘$ snakeviz benchmarking_and_profiling.cprof‘

And this was the result:


SnakeViz
Reset Root
Reset Zoom
Call Stack
~:0(<built-in method builtins.exec>) 11.3 s
benchmarking_and_profiling.py:1(<module>) 11.3 s
benchmarking_and_profiling.py:25(calc_pure_python) 11.2 s
benchmarking_and_profiling.py:7(measure_time) 10.6 s
benchmarking_and_profiling.py:67(calculate_z_serial_purepython) 10.6 s
benchmarking_and_profiling.py:67(calculate_z_serial_purepython) 10.6 s
~:0(<built-in method builtins.abs>) 2.70 s
Search:

Style: Icicle
Depth: 10
Cutoff: 1 / 1000

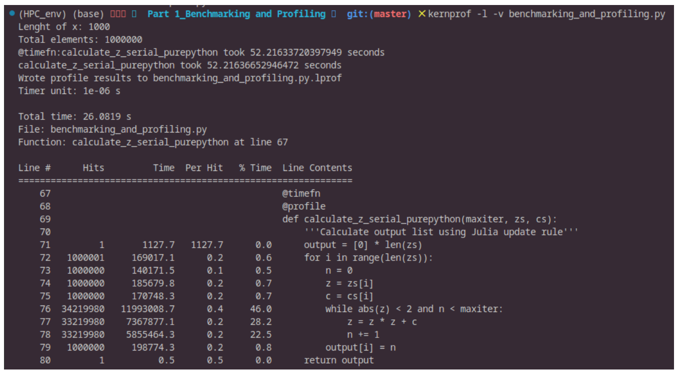
ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	7.931	7.931	10.63	10.63	benchmarking_and_profiling.py:67(calculate_z_serial_purepython)

Fig. 6. Result of snakeviz module

In this image, each block represents a function call, and the size of the block is proportional to the time spent in that function, including the cumulative time. The root block represents the total time taken by the script, which is around 11.5 seconds. The ‘run_code‘ function within IPython’s core ‘interactiveshell.py‘ module is at the top of the call stack, indicating that the profiled code was run within an IPython environment. Below, there is the ‘¡module¿‘ function from ‘benchmarking_and_profiling.py‘ with a cumulative time of 11.55 seconds. This indicates that nearly all the time taken for the script execution is within this module. Within the ‘benchmarking_and_profiling.py‘ module, the calc_pure_python function was called and has a cumulative time of 11.2 seconds, indicating that it’s the main driver of the computation time. The function ‘calculate_z_serial_purepython‘ consumes most of the execution time, with a total time of 10.6 seconds. This suggests that the function is highly CPU-intensive since is where the core computation logic for generating the Julia set is located. Also, it can be seen on the lilac block of the end the ‘abs‘ function, that took a total of 2.7 seconds.

I used the ‘line_profiler‘ and ‘kernprof‘ module to profile line-by-line the function ‘calculate_z_serial_purepython‘. To run it correctly I used:

‘$ kernprof -l -v benchmarking_and_profiling.py‘



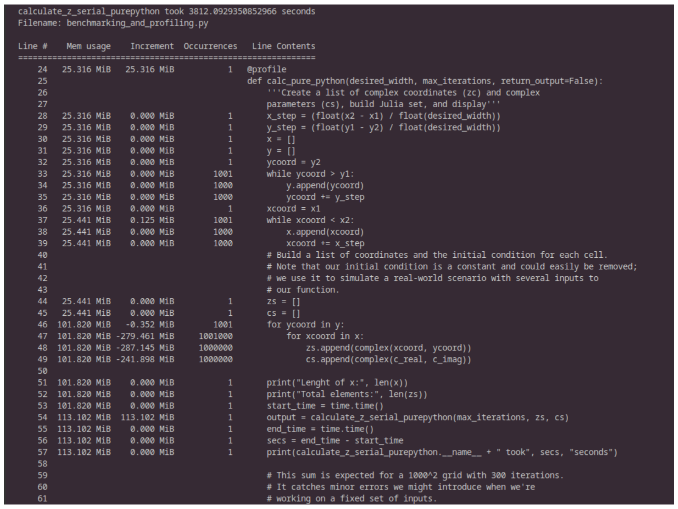
```
(HPC_env) (base) Part 1_Benchmarking and Profiling git:(master) ✗kernprof -l -v benchmarking_and_profiling.py
Lenght of x: 1000
Total elements: 1000000
@timefn:calculate_z_serial_purepython took 52.21633720397949 seconds
calculate_z_serial_purepython took 52.21636652946472 seconds
Wrote profile results to benchmarking_and_profiling.py.lprof
Timer unit: 1e-06 s

Total time: 26.0819 s
File: benchmarking_and_profiling.py
Function: calculate_z_serial_purepython at line 67

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    67                                           @timefn
    68                                           @profile
    69                                           def calculate_z_serial_purepython(maxiter, zs, cs):
    70                                               '''Calculate output list using Julia update rule'''
    71         1       1127.7   1127.7      0.0      output = [0] * len(zs)
    72   1000001     169017.1      0.2      0.6      for i in range(len(zs)):
    73   1000000     140171.5      0.1      0.5          n = 0
    74   1000000     185679.8      0.2      0.7          z = zs[i]
    75   1000000     170748.3      0.2      0.7          c = cs[i]
    76  34219980   11993008.7      0.4     46.0          while abs(z) < 2 and n < maxiter:
    77  33219980    7367877.1      0.2     28.2              z = z * z + c
    78  33219980    5855464.3      0.2     22.5              n += 1
    79   1000000     198774.3      0.2      0.8          output[i] = n
    80         1          0.5      0.5      0.0      return output
```

Fig. 7. Result of line_profiler module in calculate_z_serial_purepython

Analyzing this results, it can be seen that this function had a total run time of 26.08 seconds. Line 76, which contains the while loop was hitted 34219980 times, which consumes 46% of the total time within the function, this makes sense because is where the core computation for the set occurs. The timeper hit is small for all lines, but because the function is called many times, the cumulative time then becomes significant. From this, we can conclude that teh function is called many times, which makes that even operations that normally take small amount of time per call can add up to a significant portion of the total run time.

Lastly, I used ‘memory_profiler‘ on two functions to diagnose memory usage. To run it correctly I used:

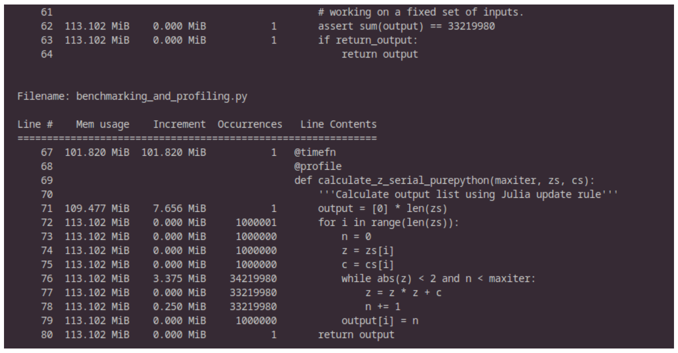
‘$ python -m memory_profiler benchmarking_and _profiling.py‘



```
calculate_z_serial_purepython took 3812.0929350852966 seconds
Filename: benchmarking_and_profiling.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    24   25.316 MiB   25.316 MiB           1   @profile
    25                                         def calc_pure_python(desired_width, max_iterations, return_output=False):
    26                                             '''Create a list of complex coordinates (zc) and complex
    27                                             parameters (cs), build Julia set, and display'''
    28   25.316 MiB    0.000 MiB           1       x_step = (float(x2 - x1) / float(desired_width))
    29   25.316 MiB    0.000 MiB           1       y_step = (float(y1 - y2) / float(desired_width))
    30   25.316 MiB    0.000 MiB           1       x = []
    31   25.316 MiB    0.000 MiB           1       y = []
    32   25.316 MiB    0.000 MiB           1       ycoord = y2
    33   25.316 MiB    0.000 MiB        1001       while ycoord > y1:
    34   25.316 MiB    0.000 MiB        1000           y.append(ycoord)
    35   25.316 MiB    0.000 MiB        1000           ycoord += y_step
    36   25.316 MiB    0.000 MiB           1       xcoord = x1
    37   25.441 MiB    0.125 MiB        1001       while xcoord < x2:
    38   25.441 MiB    0.000 MiB        1000           x.append(xcoord)
    39   25.441 MiB    0.000 MiB        1000           xcoord += x_step
    40                                             # Build a list of coordinates and the initial condition for each cell.
    41                                             # Note that our initial condition is a constant and could easily be removed;
    42                                             # we use it to simulate a real-world scenario with several inputs to
    43                                             # our function.
    44   25.441 MiB    0.000 MiB           1       zs = []
    45   25.441 MiB    0.000 MiB           1       cs = []
    46  101.820 MiB   -0.352 MiB        1001       for ycoord in y:
    47  101.820 MiB -279.461 MiB     1001000           for xcoord in x:
    48  101.820 MiB -287.145 MiB     1000000               zs.append(complex(xcoord, ycoord))
    49  101.820 MiB -241.898 MiB     1000000               cs.append(complex(c_real, c_imag))
    50
    51  101.820 MiB    0.000 MiB           1       print("Lenght of x:", len(x))
    52  101.820 MiB    0.000 MiB           1       print("Total elements:", len(zs))
    53  101.820 MiB    0.000 MiB           1       start_time = time.time()
    54  113.102 MiB  113.102 MiB           1       output = calculate_z_serial_purepython(max_iterations, zs, cs)
    55  113.102 MiB    0.000 MiB           1       end_time = time.time()
    56  113.102 MiB    0.000 MiB           1       secs = end_time - start_time
    57  113.102 MiB    0.000 MiB           1       print(calculate_z_serial_purepython.__name__ + " took", secs, "seconds")
    58
    59                                             # This sum is expected for a 1000^2 grid with 300 iterations.
    60                                             # It catches minor errors we might introduce when we're
    61                                             # working on a fixed set of inputs.
```

Fig. 8. Result of memory_profiler module in calc_pure_python

In the image it can be seen that the memory usage of calc_pure_python starts at 25.31 MiB before the function begins. As the function sets up initial variables and enters loops to populate the lists, there is no significant change in memory usage. This suggests that these lists are relatively small. A substantial memory increment occurs as the lists ‘zs‘ and ‘cs‘ are populated inside the nested loops. The memory jumps by 287.15 MiB to a total of 101.82 when creating these lists, which suggests that these lists contain a large number of complex numbers, but the largest memory increment occurs when appending ‘zs‘ and ‘cs‘ because these lists are very large. The final recorded memory usage after the function is completed, as will be seen in the next image, is 113.10 MiB, which is higher than the initial usage.



```
    61                                             # working on a fixed set of inputs.
    62  113.102 MiB    0.000 MiB           1       assert sum(output) == 33219980
    63  113.102 MiB    0.000 MiB           1       if return_output:
    64                                                 return output


Filename: benchmarking_and_profiling.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    67  101.820 MiB  101.820 MiB           1   @timefn
    68                                         @profile
    69                                         def calculate_z_serial_purepython(maxiter, zs, cs):
    70                                             '''Calculate output list using Julia update rule'''
    71  109.477 MiB    7.656 MiB           1       output = [0] * len(zs)
    72  113.102 MiB    0.000 MiB     1000001       for i in range(len(zs)):
    73  113.102 MiB    0.000 MiB     1000000           n = 0
    74  113.102 MiB    0.000 MiB     1000000           z = zs[i]
    75  113.102 MiB    0.000 MiB     1000000           c = cs[i]
    76  113.102 MiB    3.375 MiB    34219980           while abs(z) < 2 and n < maxiter:
    77  113.102 MiB    0.000 MiB    33219980               z = z * z + c
    78  113.102 MiB    0.250 MiB    33219980               n += 1
    79  113.102 MiB    0.000 MiB     1000000           output[i] = n
    80  113.102 MiB    0.000 MiB           1       return output
```

Fig. 9. Result of memory_profiler module in calculate_z_serial_purepython

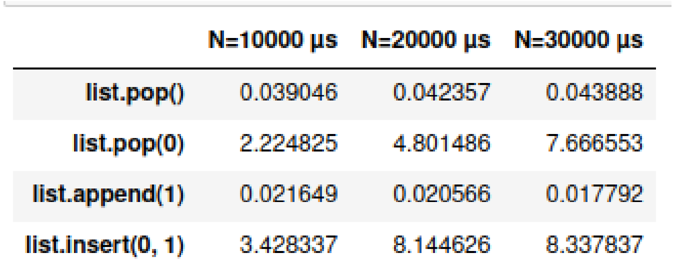
In the calculate_z_serial_purepython, the initial memory of the function is 101.82 MiB. The function increases memory usage by 7.65 MiB when it creates the list ‘output‘, this being the only significant memory increment within the function.

A. Conclusion

This exercise was useful to understand benchmarking and profiling, essential practices that can help programmers to make better and optimized functions and code with efficient resource management and that can also enhance user experience. Using this techniques, bottlenecks and inefficient resource use can be identified. Ultimately, regular use of the techniques explored can very much improve and deliver high-quality programs.

II. Part II. List and Tuples

In some cases, it is necessary to efficiently perform insertion or removal of elements both at the beginning and at the end of the collection. In this activity I will measure the time for the following operations with n = 10000, 20000 and 30000 elements. I used the ‘time‘ module to measure the time and ‘pandas‘ to make the tables. Firstly, I measured operations on list, the .pop() that deletes elements, .append() that adds elements at the end of the list, and .insert() that given an index, add an element too. This is one of the results after measuring the time with the timeit module:



	N=10000 µs	N=20000 µs	N=30000 µs
list.pop()	0.039046	0.042357	0.043888
list.pop(0)	2.224825	4.801486	7.666553
list.append(1)	0.021649	0.020566	0.017792
list.insert(0, 1)	3.428337	8.144626	8.337837

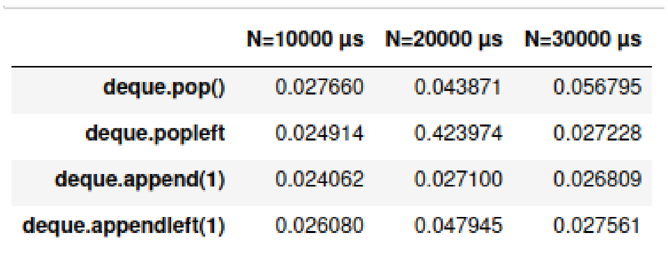
Fig. 10. Table of results using .pop(), .append(), and .insert()

Generally, removing the last item from a list ‘list.pop()‘ is a quick operation, as its time complexity is constant, and the slight fluctuation in times across different list sizes can be attributed to normal variations in a computing environment. However, removing the first item ‘list.pop(0)‘ takes considerably longer because every subsequent element must be shifted one position, a process whose time requirement grows linearly with the size of the list, this is reflected in the execution times, which increase significantly as the list gets larger.

Adding an element to the end of a list ‘list.append(1)‘ is also a constant time operation, as it is shown on the table. The slight fluctuation in times across different list sizes can be attributed to normal variations in a computing environment. Finally, inserting an element at the start of a list ‘list.insert(0, 1)‘ mirrors the behavior of ‘list.pop(0)‘ with a linear increase in time as the list grows, because all elements need to be shifted. While the execution times do increase with the list size, the rise isn’t as steep as expected for such operations, which might indicate some optimizations in the list handling by Python or other environmental factors during the test.

However, Python provides a data structure with interesting properties in the ‘collection.deque‘ class. The word deque stands for double-ended queue because this data structure is designed to efficiently put and remove elements at the beginning and at the end of the collection.

Next, evaluated ‘collection.deque‘ methods with n = 10000, 20000, and 30000.



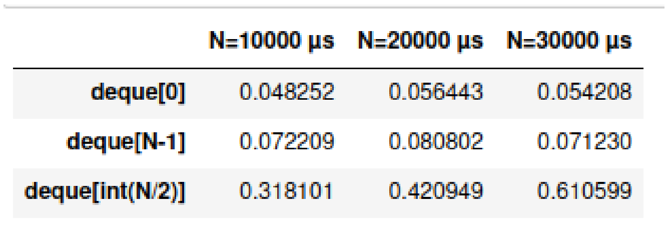
	N=10000 µs	N=20000 µs	N=30000 µs
deque.pop()	0.027660	0.043871	0.056795
deque.popleft	0.024914	0.423974	0.027228
deque.append(1)	0.024062	0.027100	0.026809
deque.appendleft(1)	0.026080	0.047945	0.027561

Fig. 11. Table of results using ‘collection.deque‘ methods

For ‘deque.pop()‘ and ‘deque.popleft()‘, which remove an element from the end and the beginning of the deque respectively, the results show only small variations in time, meaning that the time it takes to complete the task is not caused by the size of the deque.

The ‘deque.append(1)‘ operation, which adds an element to the end, and ‘deque.appendleft(1)‘, which adds an element to the beginning, show a similar pattern where the time either slightly decreases or remains fairly stable as the size of the deque increases. The small variations can be attributed to normal variations in a computing environment, such as caching effects, where larger deques might be benefiting from more efficient use of cache memory by the CPU, or perhaps the testing methodology might have some inconsistencies.

The efficiency gained by the deque.appendleft() and deque.popleft() comes at a cost: accesing an element in the middle of a deque is a O(N) operation. I evaluated the time for the next operations with N=10000, 20000, and 30000 elements to see this:



	N=10000 µs	N=20000 µs	N=30000 µs
deque[0]	0.048252	0.056443	0.054208
deque[N-1]	0.072209	0.080802	0.071230
deque[int(N/2)]	0.318101	0.420949	0.610599

Fig. 12. Table of results using ‘collection.deque‘ accesing methods

The operations on the table include accessing the first element, the last element, and an element in the middle of the deque. As the number of elements increases from 10,000 to 30,000, the time to access the first and last elements of the deque remains fairly constant and low, which aligns with the expected O(1) time complexity of these operations in a deque. However, accessing the middle element of the deque shows a different pattern. The time increases with the number of elements, which is indicative of an O(N) time complexity operation. This is because, unlike lists, deques are not optimized for random access in the middle of the structure. To access an element in the middle, a deque must iterate from one of the ends to reach that middle element, which takes linear time relative to the position of the target element.

Overallocation in lists is a memory management technique where lists are allocated with extra space beyond their current needs. This anticipates future growth, allowing for efficient additions of new items without frequent resizing of the underlying array, thereby optimizing append operations and reducing memory reallocation overhead. However, having that extra space means that a list often uses more memory than the number of its real elements need; also, in scenarios where a list doesn’t grow much after creation, the pre-allocated space may remain unused, leading to wasted memory resources, especially when dealing with a large number of lists or very large lists.

A. *Conclusion*

Throughout this activity, I’ve explored various list and deque operations and their performance implications. This has showed me that choosing between different data structures and methods can greatly impact the efficiency of a program. While lists in Python are versatile and allow for dynamic resizing through overallocation, which is particularly advantageous for frequent appends, this can lead to increased memory usage. On the other hand, deques are optimized for fast appends and pops at both ends but are not as efficient for random access operations.

The time complexity of operations like list.pop() and deque.pop() remains constant, unaffected by the size of the collection. But operations that require shifting elements, like list.pop(0) and list.insert(0, 1), have linear time complexity, with time increasing as the number of elements does. Deques provide a more performant alternative for operations at the ends, but accessing elements in the middle of a deque is costly and grows with the size of the deque.

In the end, the choice between lists and deques or even other different objects and data-types depends on the requirements of the program that will be make. If frequent insertions or deletions are needed, a deque is likely more appropriate; and if memory efficiency is more important and the collection’s size is fairly static, a list might be more suitable. Understanding the underlying behavior of these data structures allows to make informed decisions that can lead to more efficient and effective code performance.

III. Dictionaries and Sets

In this activity, I used the marco geoestadístico 2010 from [2] and 2020 [3] to obtain the ”Áreas Geoestadísticas Básicas” (AGEBs) from Mérida, Yucatán and to observe how they evolve on time using sets. To visualize the case, I will also make maps.

First, I read the data from the marco geoestadístico of 2010 using the ‘read_shp‘ function. This function loads a shapefile into a GeoDataFrame using GeoPandas, then filters the GeoDataFrame to include only those entries that belong to Mérida, identified by the 'CVEGEO' field starting with '310500001'. Then, I plotted the Áreas Geoestadísticas Básicas of 2010 using the 'plot_gdf' function, which works by checking if the 'geometry' column of the GeoDataFrame contains Shapely geometry objects. If the geometries are already in the correct format, it plots them directly. If not, it attempts to convert them into Shapely geometry objects using WKT (Well-Known Text) before plotting. And this was the result, a plot containing the AGEBs from 2010.

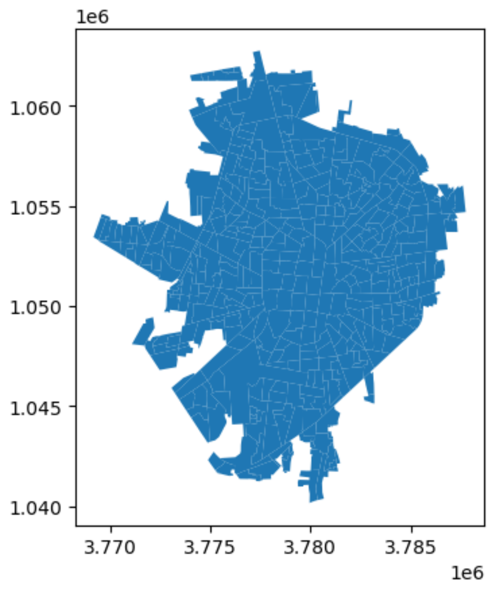


Fig. 13. AGEBs from 2010

I did the same with the data from 2020, and this was the map that resulted of that analysis.

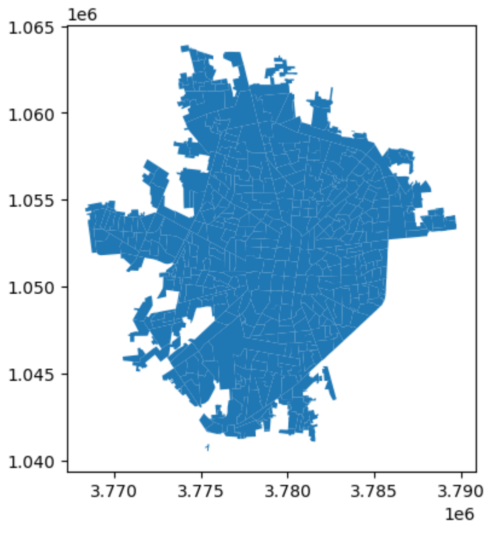


Fig. 14. AGEBs from 2020

A. AGEBs which remains from 2010 on 2020

I identified the AGEBs from the 2010 dataset that still exist in the 2020 dataset by comparing the CVEGEO. To achive this I compared these identifiers to pinpoint which AGEBs from 2010 are also present in the 2020 dataset. Sets, with their ability to handle unique elements, offer a straightforward way to perform this comparison by finding the intersection of AGEB identifiers between the two years. Once the common AGEBs are identified, I used this information to filter and visualize the geographic areas on a map. This to show the AGEBs that have persisted over the decade, offering insights into geographical continuity or change within the specified region.

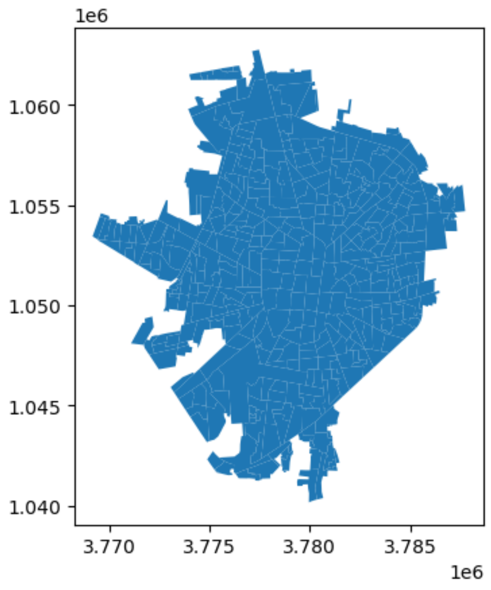


Fig. 15. AGEBs which remains from 2010 on 2020

By what I observed comparing the two GeoDataFrames, 428 AGEBs remmain after 10 years. To have a better understanding of the image, I used the 'plot_remaining_agebs', which plots the AGEBs from 2010 that are still present in 2020. The AGEBs from 2010 and 2020 are displayed as a gray background layer, and the remaining AGEBs common to both 2010 and 2020 are overlaid in purple to see the comparison better.

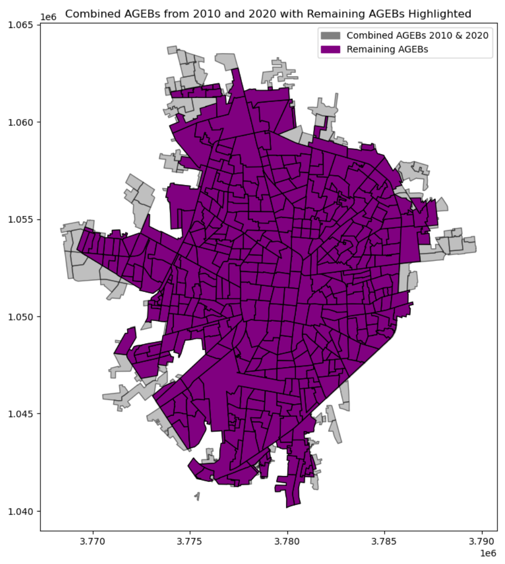

Combined AGEBs from 2010 and 2020 with Remaining AGEBs Highlighted
Combined AGEBs 2010 & 2020
Remaining AGEBs
1e6
1.065
1.060
1.055
1.050
1.045
1.040
3.770
3.775
3.780
3.785
3.790
1e6

Fig. 16. AGEBs which remains from 2010 on 2020

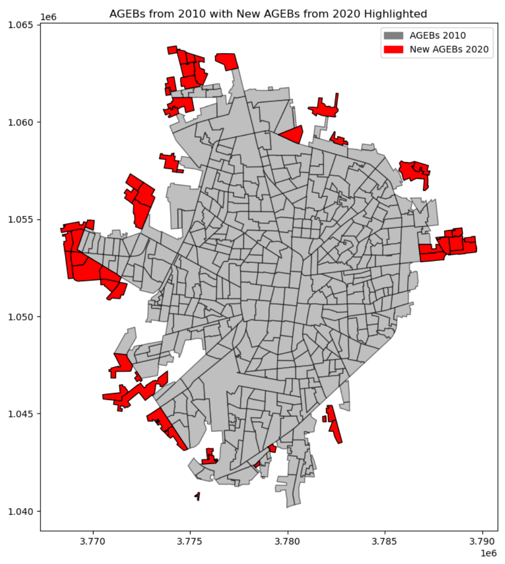


Fig. 18. AGEBs from 2010 with new AGEBs from 2020 highlighted

B. New AGEBs on 2020

In this case, the goal was to identify new AGEBs that appear in 2020 but were not present in the 2010 by examining their CVEGEO identifiers. This involved comparing the sets of identifiers from both datasets to find those that are unique to the 2020 dataset. After identifying these new AGEBs, the next step was to filter the 2020 dataset to isolate these areas. The final part of the task involved visualizing these new AGEBs on a map.

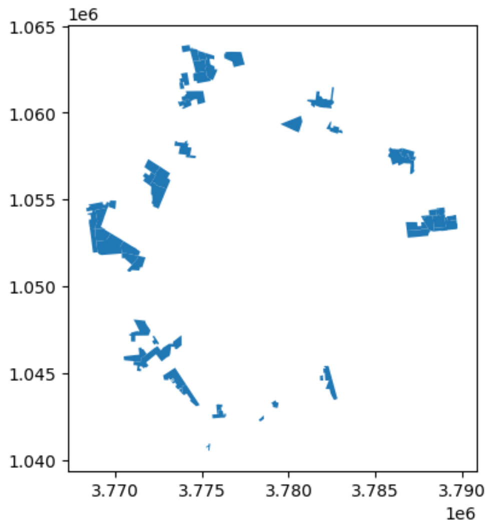


Fig. 17. New AGEBs on 2020

After analyzing the dataset with ‘len(new_agebs_gdf), it can be seen that after 10 years, 55 areas where added. I also used the ‘plot_new_agebs‘ to visualize the new AGEBs. The AGEBs from 2010 are displayed as a gray background layer, and the new AGEBs introduced in 2020 are overlaid in red.

This visualization highlights the geographic locations of the new AGEBs, and also provides insight into the spatial development and changes in Merida over the decade.

C. AGEBs that dissapear from 2010 to 2020

In this section, the objective was to identify AGEBs that were present in the 2010 dataset but are no longer found in the 2020 dataset, indicating these areas have disappeared or been redefined over the decade. This is achieved by comparing the unique CVEGEO identifiers in both datasets. Using sets to represent these identifiers from each year, the operation to find the difference will pinpoint those AGEBs that existed in 2010 but do not have a corresponding entry in the 2020 dataset. This subset represents the disappeared AGEBs. The process involves creating a filtered dataset from the 2010 data that includes only these disappeared AGEBs.

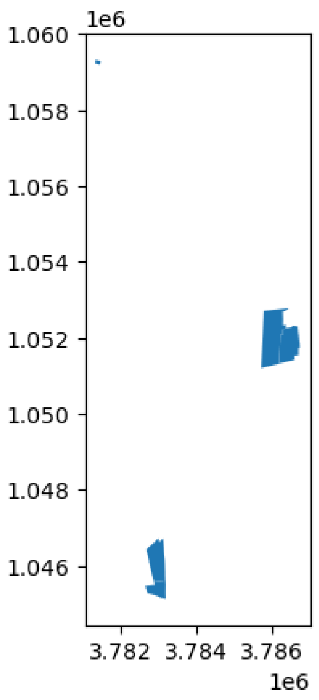


Fig. 19. AGEBs that dissapear from 2010 to 2020

After ten years, only 5 areas dissapeared or changed. I showed the comparison using ‘plot_disappeared_agebs‘ to see the change better. This function plots the AGEBs from 2010 that disappeared by 2020. The AGEBs from 2020 are displayed as a gray background layer, and the AGEBs that disappeared by 2020 are overlaid in blue.

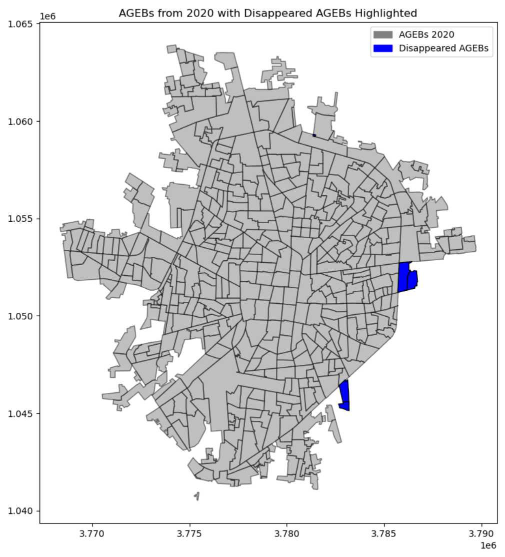


Fig. 20. AGEBs from 2020 with dissappeared AGEBs highlighted

The visual let’s us observe geographical changes and urban development patterns between 2010 and 2020, highlighting areas that may have undergone significant transformation or no longer exist as they once did.

I used another function to visualize better and more clearly the change over ten years. This function is ‘comparison_plot‘ that plots and compares the AGEBs from 2010 and 2020 GeoDataFrames on the same map for visual comparison. The function plots the 2010 AGEBs in blue and the 2020 AGEBs in red. Areas where AGEBs overlap will appear in a blended color, indicating no change. Areas in red or blue only indicate AGEBs that are new in 2020 or were only present in 2010, respectively.

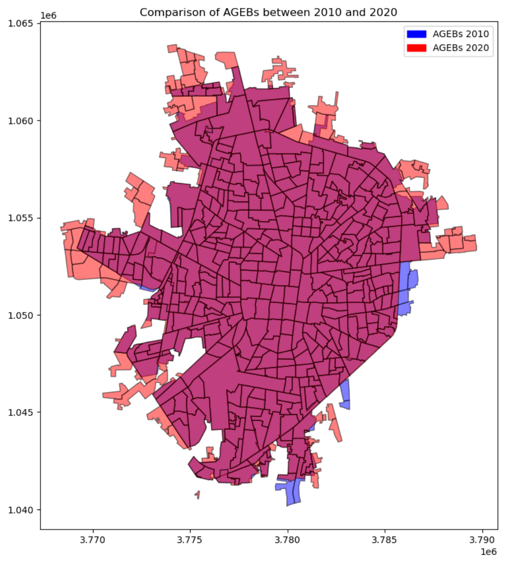


Fig. 21. Comparison of AGEBs between 2010 and 2020

D. *Conclusion*

In this activity, the evolution of ”Áreas Geoestadísticas Básicas” (AGEBs) in Mérida, Yucatán from 2010 to 2020 was explored using the marco geoestadístico datasets. Key to our analysis were sets and dictionaries, which facilitated efficient comparison of ’CVEGEO’ identifiers to identify persistent, new, and disappeared AGEBs. Sets allowed for straightforward identification of common and unique identifiers through set operations, simplifying the data handling process. The visualization of AGEB changes provided clear insights into geographical continuity and changes over the decade, aiding in the understanding of urban development patterns.

IV. Matrix and Vector Computations

After reading about Broadcasting with Arrays on the Chapter *Computation on Arrays: Broadcasting* from Python Data Science Handbook [4], I obtained the following information:

The text talks about the concept of broadcasting in NumPy, a powerful feature that allows for vectorized operations across arrays of different shapes without the need for explicit replication of data. Broadcasting simplifies code and enhances performance by enabling binary operations like addition, subtraction, multiplication, etc., to be efficiently executed between arrays of different sizes.

Broadcasting includes rules, for example, NumPY follows a set of rules to determine how operations between mismatched shapes are handled. It first pads the shape of the smaller array with ones on its leading side, then stretches dimensions of size 1 to match the corresponding dimensions of the other array. If mismatched sizes are found and neither is 1, an error is raised. It also avoids the explicit data duplication that would be necessary for operations between arrays of different shapes, leading to memory and computational efficiency. This is crucial for high-performance computing where resources and computation time are critical. These rules apply to any binary universal functions (ufuncs) in NumPy, not just addition or multiplication, making it a versatile tool for a wide range of mathematical operations. In the text there are showed various examples, including adding a scalar to an array, adding a one-dimensional array to a two-dimensional one, and more complex cases. These examples demonstrate how broadcasting supports operations like centering an array by subtracting the mean or computing functions over a grid, which are common in data processing, scientific computing, and machine learning.

In the context of High Performance Computing (HPC) and matrix/vector computations, broadcasting is useful because it allows for the efficient execution of vectorized operations, which are key to achieving high performance in numerical computations. This efficiency is due to the avoidance of explicit loops in Python, leveraging instead the optimized implementations of these operations in lower-level languages. It also simplifies the code for matrix and vector operations, making it easier to read, write, and maintain. Additionally, the ability to perform operations on arrays of different shapes and sizes without explicit reshaping or resizing is a significant advantage, because it offers flexibility in designing algorithms and implementing mathematical models.

After reading *Rewriting the particle simulator in NumPy* on ***Chapter 3: Fast Array Operations with NumPy and Pandas (pp. 68)*** from [5]. I will implement the improvements on the particle simulator using NumPy and show that both implementations scale linearly with particle size, but the runtime in the pure Python version grows much faster than the NumPy version.

The ‘evolve_python‘ method of the ‘ParticleSimulator‘ class uses pure Python loops and arithmetic operations to compute the evolution of particles over time. It calculates the norm and the velocity components for each particle individually within a nested loop structure. This can be computationally inefficient due to Python’s inherent overhead in looping and handling individual arithmetic operations in the interpreted runtime environment. On the other hand, the ‘evolve_numpy‘ method takes advantage of NumPy’s array operations. This allows the method to compute the norms and velocity components for all particles at once. The operations are applied to whole arrays of data in parallel, making use of lower-level optimizations and potentially faster CPU instructions.

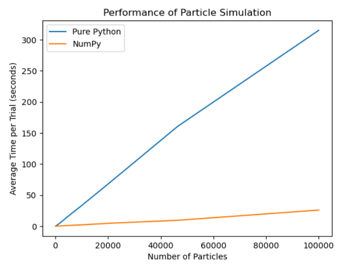


Fig. 22. Performance comparison of the Particle Simulator program

The graph illustrates the comparative performance of the two particle simulation methods as the number of particles increases. The execution time for both methods rises in direct proportion to the number of particles, which indicates that the complexity of both algorithms is linear, each additional particle seems to add a roughly constant amount of time to the simulation. However, the pure Python version’s execution time increases at a much steeper rate than that of the NumPy version. This demonstrates the higher efficiency of NumPy for numerical computations, especially as the number of particles grows. With a larger dataset, NumPy’s advantage becomes more pronounced, emphasizing its ability to handle extensive numerical operations more effectively.

Now, I’ll explain how to obtain the optimal performance with ‘numexpr‘. To do this, first, I read the section **Reaching optimal performance with numexpr, pp. 72** of [5]. I will implement it and measure the execution time.

The usage of ‘numexpr‘ is generally straightforward and based on a single function: ‘numexpr.evaluate‘. The ‘numexp‘ package increases the performances in almost all cases, but to get a substantial advantage, you should use it with large arrays. For example, an application that involves a large array is the calculation of a distance matrix. In a particle system, a distance matrix contains all the possible distances between the particles. To calculate it, we should first calculate all the vectors connecting any two particles (i,j). Then, we calculate the length of this vector by taking its norm. To compare, I wrote two implementations, one with ‘numpy‘ and one with ‘numexpr‘ and compared its performance.

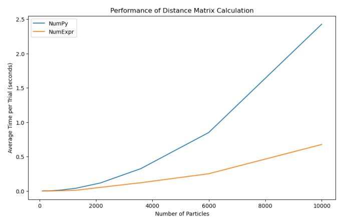


Fig. 23. Performance of distance matrix calculation

After this implementation, it can be seen that using the ‘numexpr‘ package improves performance. This enhancement occurs because of ‘numexpr‘’s ability to optimize array expressions dynamically, minimizing memory allocation for intermediate results and efficiently utilizing CPU cache. Additionally, ‘numexpr‘ leverages multiple processors, distributing computations to execute in parallel.

The key to achieving optimal performance with ‘numexpr‘ lies in its application to large arrays and complex expressions. By packing as much of the computation as possible into single ‘numexpr.evaluate‘ calls, you reduce the overhead of Python’s interpretation cycle and make full use of ‘numexpr‘’s optimizations.

A. Conclusion

The integration of NumPy and NumExpr into numerical computations represents a powerful strategy for enhancing performance. NumPy’s broadcasting facilitates the efficient manipulation of arrays, while NumExpr’s optimized expression evaluation further accelerates computation. These tools can offer optimal performance in Python’s numerical and array-based operations, making them indispensable for applications requiring high efficiency and scalability.

V. Compiling to C

Conway’s Game of Life is a classic example of a cellular automaton, first devised by the British mathematician John Horton Conway in 1970. This zero-player game unfolds on an infinite two-dimensional grid of cells, each of which can be in one of two states: alive (1) or dead (0). The evolution of the grid is determined by a set of rules that simulate the birth, death, and survival of cells.

The game progresses in discrete time steps, or generations. At each step, the state of every cell is updated simultaneously based on the states of its eight neighboring cells.

- An alive cell will remain alive in the next generation if it has two or three living neighbors, reflecting survival.
- An alive cell will die if it has fewer than two or more than three living neighbors.
- A dead cell will become alive if it has exactly three living neighbors, representing reproduction.
- In all other cases, a dead cell remains dead.

While the Game of Life is conceptually straightforward and can be implemented in any programming language, its computational performance can vary dramatically depending on the approach used. This is where the journey from Python to C through Cython becomes particularly interesting. Python, known for its readability and ease of use, is not always the most efficient language for processing intensive computational tasks such as simulating the Game of Life over large grids or for many generations. This is primarily due to its high-level nature and the overhead of dynamic typing.

To address these performance concerns, I explored five different implementations of the Game of Life: one in Python and four progressively optimized versions using Cython.

In the Python version, the implementation uses the ‘update‘ function, that computes the next generation of the lattice (grid), following the rules of the Game of Life. And the ‘update_rule‘ function determines the fate of a single cell based on its neighbors’ states. The ‘initialize_lattice_with_cross‘ function creates an initial grid (lattice) with a specific size (‘box_size‘), and initializes it to have a ”cross” pattern of living cells at the center. Together, these functions set up the initial state of the grid and then evolve it by one generation.

The functions of the second version uses Cython. The code improves primarily the compilation process. Cython compiles Python code to C, which is then compiled to machine code, leading to faster execution times. In this case, the nested for-loops in update, which iterate over each cell in the lattice, run faster because compiled C loops are more efficient than their interpreted Python counterparts. Also, the function ‘c1_update_rule‘ is called repeatedly within these loops. In Python, function calls are expensive, but in Cython, they can be optimized, especially if type declarations are used. Lastly, since the code is running at the C level, the overhead of Python’s dynamic type checking and other runtime checks are avoided, resulting in performance gains. In this version, changes were not made to the logic of the code; the improvement is result of the Cython compilation process and the efficiencies it brings to Python code execution.

In the second Cython implementation, the code has been adapted for Cython by adding type declarations to variables using ‘cdef‘. These type declarations inform Cython that the variables are integers. This allows Cython to convert these Python variables into C variables when it compiles the code to C. By knowing the types beforehand, Cython can generate more optimized C code. This is because C is a statically-typed language, and when the types are known at compile time, the compiler can optimize the memory usage and the machine code it generates for operations involving those variables.

The third version of Cython uses the ‘cdef‘ keyword in front of the ‘c3_update_rule‘ function declaration, indicating that the function is now a Cython function rather than a Python function. This change enhances performance further because Cython can now treat the function as a C function. As a result, when ‘c3_update_rule‘ is called from within the ‘c3_update‘ function, the call is made at C speed without the overhead of Python’s function-calling mechanisms.

Lastly, the fourth implementation of Cython uses Cython compiler directives that instruct the Cython compiler to disable certain checks that are normally in place for safety but can slow down execution. By setting ‘boundscheck=False‘, Cython skips checking whether index operations are within the bounds of the array or list. Similarly, wraparound=False disables the ability to use negative indices to access arrays from the end. These checks are useful and necessary for preventing errors in development and ensuring correctness in the code. However, they do incur a performance penalty due to the extra overhead. In tight loops, especially like those found in ‘c4_update‘, which may be executed millions of times for large lattices, these checks can significantly slow down execution. This implementation is particularly useful in scenarios where the code has been thoroughly tested, and it is certain that index errors will not occur. This typically comes into play with well-understood algorithms like the Game of Life, where the boundaries and indexing patterns are predictable after careful development and testing.

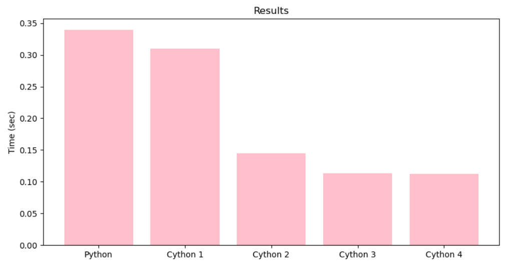


Fig. 24. Results

A. Conclusion

The progression from Python to increasingly refined Cython code demonstrates a consistent reduction in execution time, as shown in the tests. With each code implementation, strategic enhancements were integrated, such as type declarations and the removal of safety checks, resulting in a leaner and more efficient execution path that maximizes the hardware's potential.

The trend is evident; as the code moves from the flexibility and simplicity of Python to the rigidity and raw speed of C, the performance benefits become clear. The initial transition to Cython offers a modest boost, but as the implementations dive deeper into Cython's capabilities, exploiting its ability to act as a bridge between Python's expressiveness and C's performance, we see that the execution time continues to drop. The culmination of this process in the fourth Cython implementation is the most performant, showcasing that for computationally intensive applications the careful application of Cython's features can lead to significant speed enhancements.

References

[1] M. Gorelick and I. Ozsvald, High performance Python: Practical Performant Programming for Humans. O'Reilly Media, 2020.

[2] INEGI, "Marco Geoestadístico 2010 versión 5.0 (Censo de Población y Vivienda 2010)." INEGI, 2010. [Online]. Available: https://www.inegi.org.mx/app/biblioteca/ficha.html?upc=702825292812

[3] INEGI, "Marco Geoestadístico. Censo de Población y Vivienda 2020." INEGI, 2020. [Online]. Available: https://www.inegi.org.mx/app/biblioteca/ficha.html?upc=889463807469

[4] J. VanderPlas, "Computation on Arrays: Broadcasting — Python Data Science Handbook," GItHub, Nov. 2016. https://jakevdp.github.io/PythonDataScienceHandbook/02.05-computation-on-arrays-broadcasting.html (accessed Mar. 28, 2024).

[5] G. Lanaro, Python High Performance, 2nd ed. Packt Publishing Ltd, 2017.