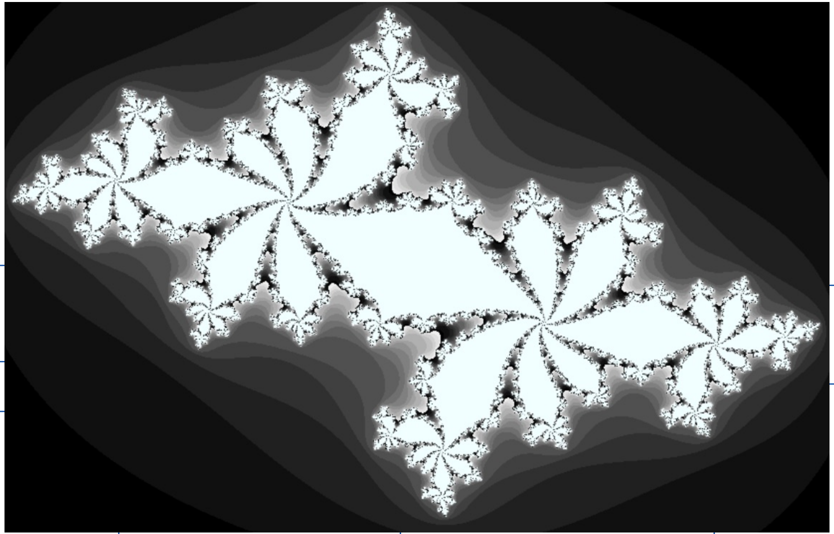
DD2358 – Timing & Julia Set Code

Stefano Markidis

KTH Royal Institute of Technology



Intended Learning Outcomes

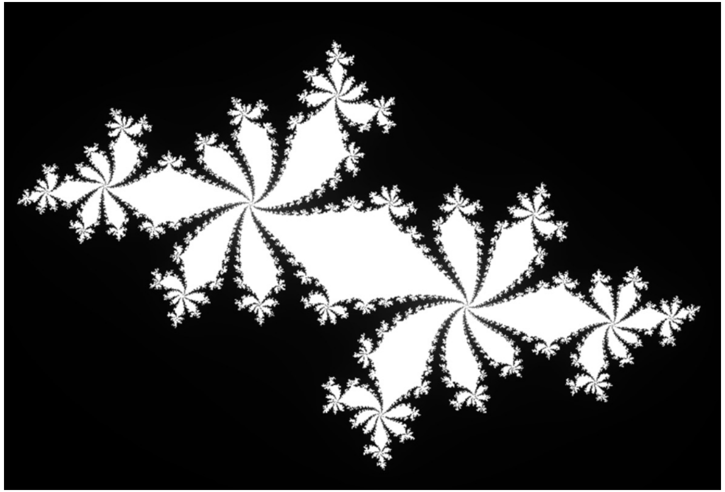
- To time different parts of our Python code using different approaches
- To write a Python decorator to wrap a function with timers

Example Python Code: Calculate the Julia Set

- We use the Julia Set code as example to showcase how to use timers and profilers.
- This code and configuration 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.

Basic Algorithm

- Each pixel is a complex number:
  - Real part is the x-coordinate
  - Imaginary part is y-coordinate
- We have a loop for going through all the pixels
- On each pixel, we apply a function to calculate new values and we test the results
  - 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*
    - > *Those that cause a high number of iterations are colored white.*
      - **Computational imbalance:** some pixel will take more time than other ones



Basic Algorithm - II

- We define a set of z coordinates that we'll test. The function that we **calculate squares the complex number `z` and sum `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`, we break out of the loop and record the number of iterations we performed at this coordinate.
- If the escape function is never `False`, we stop after `maxiter` iterations.
  - We will later turn this z's result into a colored pixel representing this complex location.

```
for z in coordinates:
    for iteration in range(maxiter):  # limited iterations per pc
        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
```

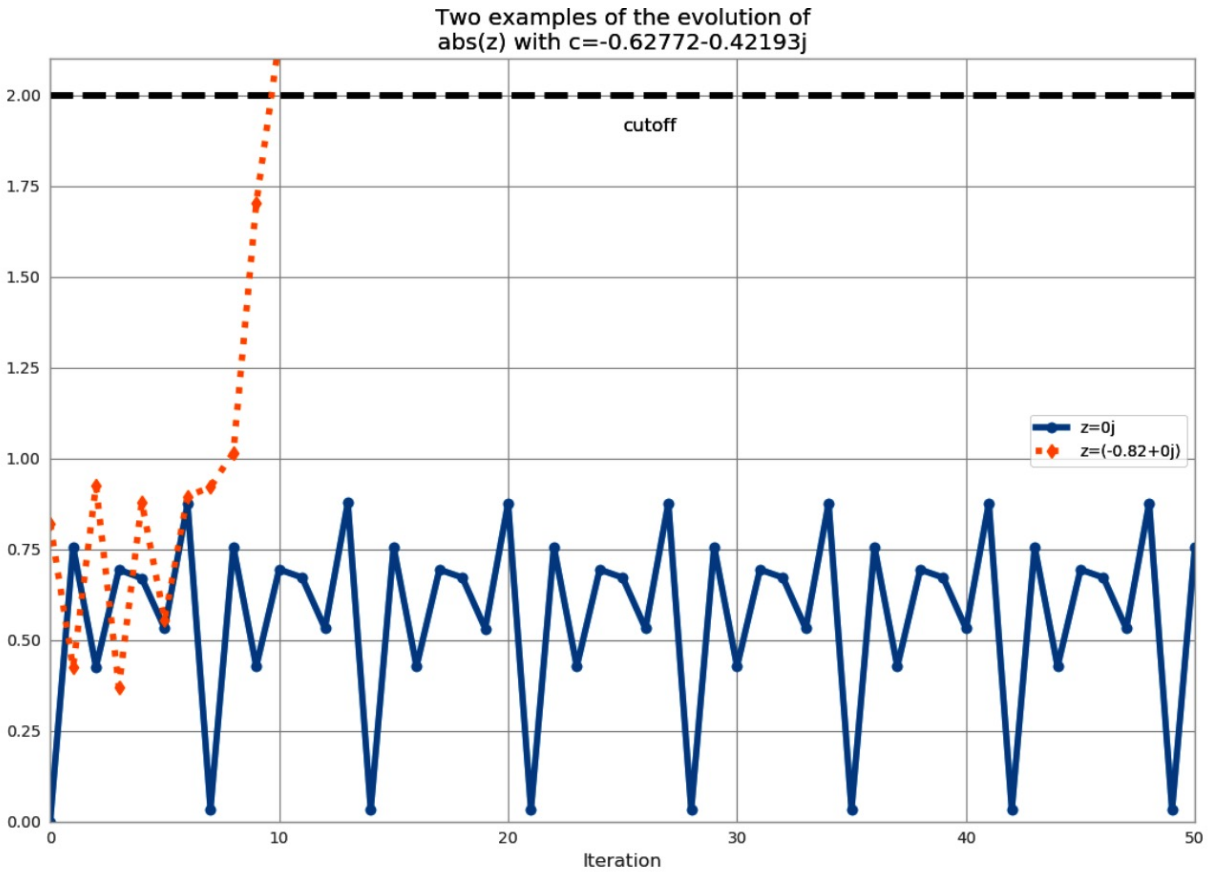
Example – One Pixel

```python
c = -0.62772-0.42193j
z = 0+0j
for n in range(9):
    z = z*z + c
    print(f"{n}: z={z: .5f}, abs(z)={abs(z):0.3f}, c={c: .5f}")
```

```
0: z=-0.62772-0.42193j, abs(z)=0.756, c=-0.62772-0.42193j
1: z=-0.41171+0.10778j, abs(z)=0.426, c=-0.62772-0.42193j
2: z=-0.46983-0.51068j, abs(z)=0.694, c=-0.62772-0.42193j
3: z=-0.66777+0.05793j, abs(z)=0.670, c=-0.62772-0.42193j
4: z=-0.18516-0.49930j, abs(z)=0.533, c=-0.62772-0.42193j
5: z=-0.84274-0.23703j, abs(z)=0.875, c=-0.62772-0.42193j
6: z= 0.02630-0.02242j, abs(z)=0.035, c=-0.62772-0.42193j
7: z=-0.62753-0.42311j, abs(z)=0.757, c=-0.62772-0.42193j
8: z=-0.41295+0.10910j, abs(z)=0.427, c=-0.62772-0.42193j
```

- Each update to `z` for these first iterations leaves it with a value where `abs(z) < 2` is `True`.
- For this coordinate we can iterate 300 times, and still the test will be `True`.
  - We cannot tell how many iterations we must perform before the condition becomes `False`, and this may be an infinite sequence.
  - The maximum iteration (maxiter) break clause will stop us from iterating potentially forever.

Example – Two Pixels Evolution – 50 Iterations



Calculating the Full Julia Set - Timing

- We break down the code that generates the Julia set.
- At the start of our module, we import the time module for our first profiling approach and define **some coordinate constants**.

```
import time


# area of complex space to investigate
x1, x2, y1, y2 = -1.8, 1.8, -1.8, 1.8
c_real, c_imag = -0.62772, -.42193
```

```python
def calc_pure_python(desired_width, max_iterations):
    """Create a list of complex coordinates (zs) and complex p
    build Julia set"""
    x_step = (x2 - x1) / desired_width
    y_step = (y1 - y2) / desired_width
    x = []
    y = []
    ycoord = y2
    while ycoord > y1:
        y.append(ycoord)
        ycoord += y_step
    xcoord = x1
    while xcoord < x2:
        x.append(xcoord)
        xcoord += x_step
    # build a list of coordinates and the initial condition fo
    # Note that our initial condition is a constant and could 
    # we use it to simulate a real-world scenario with several
    # function
    zs = []
    cs = []
    for ycoord in y:
        for xcoord in x:
            zs.append(complex(xcoord, ycoord))
            cs.append(complex(c_real, c_imag))

    print("Length of x:", len(x))
    print("Total elements:", len(zs))
```

Lists

Creating Lists

- We create two lists of input data.
  - The first is `zs` (complex *z* coordinates)
  - The second is `cs` (a complex initial condition)
- To build the `zs` and `cs` lists, we need to know the coordinates for each z.
  - We build up these coordinates using `xcoord` and `ycoord` and a specified `x_step` and `y_step`.

Function with Computation

```
def calculate_z_serial_purepython(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 define the `calculate_z_serial_purepython` function, which expands on the algorithm defined before.
- We also define an output list at the start that has the same length as the input `zs` and `cs` lists.

Timing with `time.time()`

```python
def calc_pure_python(desired_width, max_iterations):
    """Create a list of complex coordinates (zs) and complex p
    build Julia set"""
    x_step = (x2 - x1) / desired_width
    y_step = (y1 - y2) / desired_width
```

…

```python
print("Length of x:", len(x))
print("Total elements:", len(zs))
start_time = time.time()
output = calculate_z_serial_purepython(max_iterations, zs, cs)
end_time = time.time()
secs = end_time - start_time
print(calculate_z_serial_purepython.__name__ + " took", secs, "

# This sum is expected for a 1000^2 grid with 300 iterations
# It ensures that our code evolves exactly as we'd intended
assert sum(output) == 33219980
```

Start timer

Stop timer

- We calculate the output list via `calculate_z_serial_purepython`
- We use `time.time()` to retrieve timing
  - function returns the **number of seconds** passed since epoch.
    - For Unix system, *January 1, 1970, 00:00:00* at **UTC** is epoch (the point where time begins).
- We sum the contents of output and assert that it matches the expected output value

Main Code

- Now we call the calculation.
- By wrapping it in a `__main__` check, we can safely import the module without starting the calculations for some of the profiling methods.

```
if __name__ == "__main__":
    # Calculate the Julia set using a pure Python solution witl
    # reasonable defaults for a laptop
    calc_pure_python(desired_width=1000, max_iterations=300)
```

Running the Code…

```
# running the above produces:
Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 8.087012767791748 seconds
```

When reporting timing information, always:

- Run more the once (timing gives non-deterministic results)
- Report average and standard deviation of the timing

Simple Approaches to Timing - Print and a Decorator

- Using print statements is commonplace when debugging and profiling code.
  - It quickly becomes unmanageable.
    - > *We tidy up the print statements when you're done with them*
- A cleaner approach is to use a Python *decorator*
  - here, we add one line of code above the function that we care about.

A Timing Decorator - I

```
from functools import wraps

def timefn(fn):
    @wraps(fn)
    def measure_time(*args, **kwargs):
        t1 = time.time()
        result = fn(*args, **kwargs)
        t2 = time.time()
        print(f"@timefn: {fn.__name__} took {t2 - t1} seconds")
        return result
    return measure_time

@timefn
def calculate_z_serial_purepython(maxiter, zs, cs):
    ...
```

Our timing decorator

- We define a new function, `timefn` take **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 execution.
- Around the execution of `fn`, we capture `time.time()` and then print the result along with `fn.__name__`.

A Timing Decorator - II

```python
from functools import wraps

def timefn(fn):
    @wraps(fn)
    def measure_time(*args, **kwargs):
        t1 = time.time()
        result = fn(*args, **kwargs)
        t2 = time.time()
        print(f"@timefn: {fn.__name__} took {t2 - t1} seconds")
        return result
    return measure_time

@timefn
def calculate_z_serial_purepython(maxiter, zs, cs):
    ...
```

– We use `@wraps(fn)` from `functools` to expose the function name and docstring to the caller of the decorated function (copying attributes

> *Otherwise, we would see the function name and docstring for the decorator, not the function it decorates).*

Running with the Decorator

```python
from functools import wraps

def timefn(fn):
    @wraps(fn)
    def measure_time(*args, **kwargs):
        t1 = time.time()
        result = fn(*args, **kwargs)
        t2 = time.time()
        print(f"@timefn: {fn.__name__} took {t2 - t1} seconds"
        return result
    return measure_time

@timefn
def calculate_z_serial_purepython(maxiter, zs, cs):
    ...
```

Add the decorator before function the definition

```
Length of x: 1000
Total elements: 1000000
@timefn:calculate_z_serial_purepython took 8.00485110282898 sec
calculate_z_serial_purepython took 8.004898071289062 seconds
```

`timeit` for Coarse Measurements

- From the command line, we can run `timeit` as follows:

```
python -m timeit -n 5 -r 1 -s "import JuliaSet" \
  "JuliaSet.calc_pure_python(desired_width=1000, max_iterations=300)”
```

- We specify the number of loops `(-n 5)` and the number of repetitions `(-r 1)` to repeat the experiments.

timeit

```
stef@Stefs-MacBook-Air Codes % python -m timeit -n 5 -r 1 -s "import JuliaSet" \
 "JuliaSet.calc_pure_python(desired_width=1000, max_iterations=300)"
1 Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 3.907364845275879 seconds
@timefn: calc_pure_python took 4.156301021575928 seconds
2 Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 3.8968217372894287 seconds
@timefn: calc_pure_python took 4.136440992355347 seconds
3 Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 3.8983099460601807 seconds
@timefn: calc_pure_python took 4.131953954696655 seconds
4 Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 3.9007961750030518 seconds
@timefn: calc_pure_python took 4.138447999954224 seconds
5 Length of x: 1000
Total elements: 1000000
calculate_z_serial_purepython took 3.899070978164673 seconds
@timefn: calc_pure_python took 4.137251853942871 seconds
5 loops, best of 1: 4.14 sec per loop
```

Simple Timing using Unix `time` command

- The following will record various views on the execution time of your program, and it won't care about the internal structure of your code

Important

- **/usr/bin/**`time python JuliaSet.py`
- We get three results:
  - `real` records the wall clock or elapsed time.
  - `user` records the amount of time the CPU spent on our task outside of kernel functions.
  - `sys` records the time spent in kernel-level functions.
- By adding `user` and `sys`, you get a sense of how much time was spent in the CPU.
  - The difference between this and `real` might tell us about the amount of time spent waiting for I/O; it might also suggest that your system is busy running other tasks that are distorting your measurements.

To Summarize

- I introduced a Python example code that we will use to experiment with different profilers and timers – Julia set code
- We can use `time.time()` method to start and stop timers
- Timing is a non-deterministic measurements so we need to repeat timing and report average and standard deviation
- For timing purposes, it cleaner to use a timing decorator (a function that takes a function as argument) to wrap timers around function
- We can use the `timit` module to timing from the command line
- Use /usr/bin/time utility for coarse-grained measurements