E1
T1.1
ProductName: macOS
ProductVersion: 15.0.1
BuildVersion: 24A348
Darwin Mac 24.0.0 Darwin Kernel Version 24.0.0: Tue Sep 24 23:37:13 PDT 2024; root:xnu-11215.1.12~1/RELEASE_ARM64_T8112 arm64

time.time: 7.152557373046875e-07
timeit.default_timer: 8.297502063214779e-08
time.time_ns: 768.0

ProductName: macOS
ProductVersion: 14.6.1
BuildVersion: 23G93
Darwin Mac 23.6.0 Darwin Kernel Version 23.6.0: Mon Jul 29 21:14:30 PDT 2024; root:xnu-10063.141.2~1/RELEASE_ARM64_T6030 arm64
time.time: 7.152557373046875e-07
timeit.default_timer: 8.288770914077759e-08
time.time_ns: 768.0

clock_granularity.py was used to test this which is located in A1/E1/T1_1/clock_granularity.py

T1.2
We ran the command:

```
“python3 -m timeit -n 5 -r 1 -s "import JuliaSet"
"JuliaSet.calc_pure_python(desired_width=1000,
max_iterations=300)"
```

With the JuliaSet.py using our own made decorator.py in A1/E1/T1_2/decorator.py

Results for computer with core speed of 4.05 GHz (assumed):
Function: calculate_z_serial_purepython
  Average Execution Time: 2.619 seconds
  Average Standard Deviation: 0.0103 seconds
Function: calc_pure_python
  Average Execution Time: 2.811 seconds
  Average Standard Deviation: 0.0215 seconds

With a clock frequency of about 4.05 GHz, a single cycle is about 0.25ns which is vastly smaller than both standard deviations (as they have 10.3 ms and 21.5 ms respectively). This means the amount of cycles must differ between each run or other programmes take up cycles on the system when running the code.

It might as well be the case that the OS decides to not use the performance cores in the CPU. These have a lower clock frequency of 2.75 GHz which results in the per cycle time of

0.36ns. This is still magnitudes smaller than the standard deviation of 10.3 ms and 21.5 ms of the average standard deviations. The kernel thread scheduling could allocate varying amounts of CPU time to each thread, contributing to the observed variation in execution times.

T1.3

Using the cprofile command:

```
python3 -m cProfile -s cumulative JuliaSet.py
```

The top results with cProfile:

```
  Ordered by: cumulative time

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   118/1    0.003    0.000    5.366    5.366 {built-in method builtins.exec}
       1    0.012    0.012    5.366    5.366 JuliaSet.py:1(<module>)
       1    0.327    0.327    5.189    5.189 JuliaSet.py:27(calc_pure_python)
       1    3.325    3.325    4.729    4.729 JuliaSet.py:67(calculate_z_serial_purepython)
34219980    1.404    0.000    1.404    0.000 {built-in method builtins.abs}
      14    0.001    0.000    0.403    0.029 __init__.py:1(<module>)
   157/1    0.001    0.000    0.165    0.165 <frozen importlib._bootstrap>:1349(_find_and_load)
   157/1    0.000    0.000    0.165    0.165 <frozen importlib._bootstrap>:1304(_find_and_load_unlocked)
   382/3    0.000    0.000    0.165    0.055 <frozen importlib._bootstrap>:480(_call_with_frames_removed)
   145/3    0.000    0.000    0.165    0.055 <frozen importlib._bootstrap>:911(_load_unlocked)
   116/1    0.000    0.000    0.165    0.165 <frozen importlib._bootstrap_external>:989(exec_module)
       1    0.000    0.000    0.165    0.165 decorator.py:1(<module>)
  375/10    0.000    0.000    0.159    0.016 {built-in method builtins.__import__}
  251/41    0.000    0.000    0.158    0.004 <frozen importlib._bootstrap>:1390(_handle_fromlist)
 2005342    0.132    0.000    0.132    0.000 {method 'append' of 'list' objects}
       1    0.000    0.000    0.086    0.086 __config__.py:1(<module>)
     116    0.001    0.000    0.058    0.000 <frozen importlib._bootstrap_external>:1062(get_code)
 145/141    0.000    0.000    0.037    0.000 <frozen importlib._bootstrap>:806(module_from_spec)
   25/24    0.000    0.000    0.035    0.001 <frozen importlib._bootstrap_external>:1287(create_module)
   25/24    0.021    0.001    0.035    0.001 {built-in method _imp.create_dynamic}
     156    0.001    0.000    0.033    0.000 <frozen importlib._bootstrap>:1240(_find_spec)
       1    0.000    0.000    0.029    0.029 numerictypes.py:1(<module>)
     116    0.000    0.000    0.028    0.000 <frozen importlib._bootstrap_external>:1183(get_data)
       1    0.000    0.000    0.027    0.027 index_tricks.py:1(<module>)
     116    0.024    0.000    0.024    0.000 {method 'read' of '_io.BufferedReader' objects}
       1    0.000    0.000    0.024    0.024 defmatrix.py:1(<module>)
       1    0.000    0.000    0.022    0.022 _type_aliases.py:1(<module>)
```

cProfile is showing results higher than the built decorator because of the fact that it has a lot more overhead done than the decorator. It also differentiates calc_pure_python from calculate_z_serial_purepython part which means it does not include it in the first function in cProfile while the decorator includes both. This makes cProfile include more overhead but is as well more informative than the decorator.

Using the command

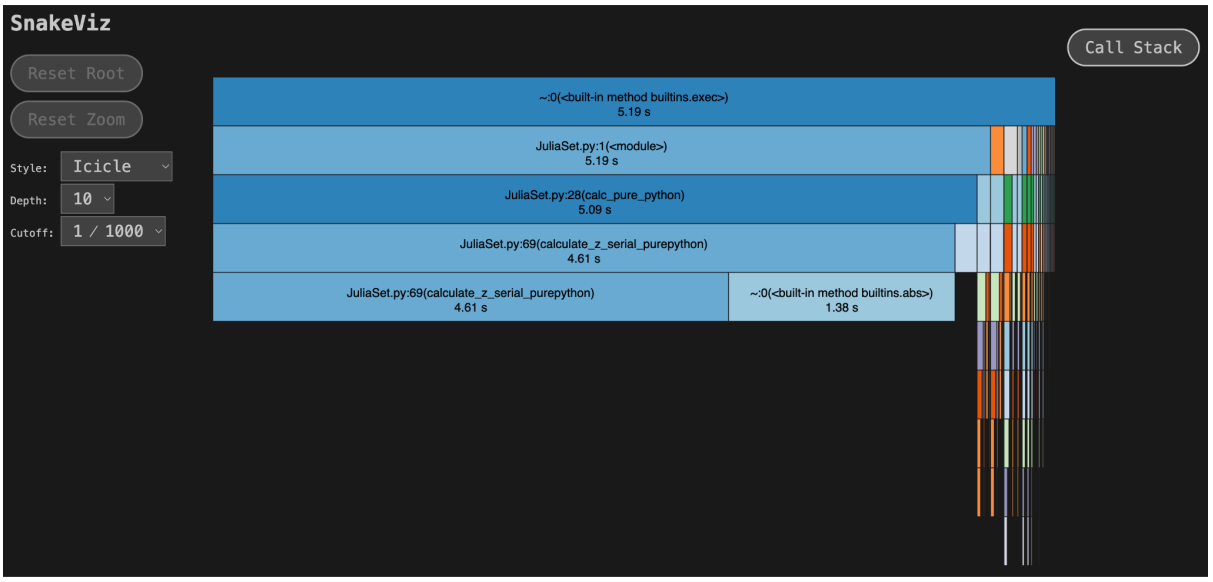
```
python3 -m kernprof -l JuliaSet.py
```

The results with line profiler becomes:

```
Timer unit: 1e-06 s

Total time: 6.04551 s
File: JuliaSet.py
Function: calc_pure_python at line 27

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    27                                           @profile
    28                                           def calc_pure_python(desired_width, max_iterations):
    29                                               """Create a list of complex coordinates (zs) and complex parameters (cs),
    30                                               build Julia set"""
    31         1          2.0      2.0      0.0      x_step = (x2 - x1) / desired_width
    32         1          1.0      1.0      0.0      y_step = (y1 - y2) / desired_width
    33         1          1.0      1.0      0.0      x = []
    34         1          5.0      5.0      0.0      y = []
    35         1          0.0      0.0      0.0      ycoord = y2
    36      1001       1118.0      1.1      0.0      while ycoord > y1:
    37      1000       1602.0      1.6      0.0          y.append(ycoord)
    38      1000       1396.0      1.4      0.0          ycoord += y_step
    39         1          4.0      4.0      0.0      xcoord = x1
    40      1001       1565.0      1.6      0.0      while xcoord < x2:
    41      1000       1564.0      1.6      0.0          x.append(xcoord)
    42      1000       1477.0      1.5      0.0          xcoord += x_step
    43                                               # build a list of coordinates and the initial condition for each cell.
    44                                               # Note that our initial condition is a constant and could easily be removed,
    45                                               # we use it to simulate a real-world scenario with several inputs to our
    46                                               # function
    47         1          0.0      0.0      0.0      zs = []
    48         1          0.0      0.0      0.0      cs = []
    49      1001        187.0      0.2      0.0      for ycoord in y:
    50   1001000     164862.0      0.2      2.7          for xcoord in x:
    51   1000000     160594.0      0.2      2.7              zs.append(complex(xcoord, ycoord))
    52   1000000     158345.0      0.2      2.6              cs.append(complex(c_real, c_imag))
    53
    54                                               # print("Length of x:", len(x))
    55                                               # print("Total elements:", len(zs))
    56         1         27.0     27.0      0.0      start_time = time.time()
    57         1    5552752.0    6e+06     91.8      output = calculate_z_serial_purepython(max_iterations, zs, cs)
    58         1          2.0      2.0      0.0      end_time = time.time()
    59         1          2.0      2.0      0.0      secs = end_time - start_time
    60                                               # print(calculate_z_serial_purepython.__name__ + " took", secs, "seconds")
    61
    62                                               # This sum is expected for a 1000^2 grid with 300 iterations
    63                                               # It ensures that our code evolves exactly as we'd intended
    64                                               # assert sum(output) == 33219980

  6.05 seconds - JuliaSet.py:27 - calc_pure_python
```

```
marcuscarlbom@MBPsomtrMarcus2 A1 % python3 -m line_profiler -rmt "JuliaSet.py.lprof"
Timer unit: 1e-06 s

Total time: 12.4048 s
File: JuliaSet.py
Function: calculate_z_serial_purepython at line 68

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    68                                           @profile
    69                                           def calculate_z_serial_purepython(maxiter, zs, cs):
    70                                               """Calculate output list using Julia update rule"""
    71         1        965.0    965.0      0.0      output = [0] * len(zs)
    72   1000001     131045.0      0.1      1.1      for i in range(len(zs)):
    73   1000000      94417.0      0.1      0.8          n = 0
    74   1000000      94801.0      0.1      0.8          z = zs[i]
    75   1000000      87967.0      0.1      0.7          c = cs[i]
    76  34219980    5457886.0      0.2     44.0          while abs(z) < 2 and n < maxiter:
    77  33219980    3467934.0      0.1     28.0              z = z * z + c
    78  33219980    2972864.0      0.1     24.0              n += 1
    79   1000000      96916.0      0.1      0.8          output[i] = n
    80         1          1.0      1.0      0.0      return output

 12.40 seconds - JuliaSet.py:68 - calculate_z_serial_purepython
```

Snakeviz visualisation



Measurements of overhead:

Running JuliaSet.py:

```
marcuscarlbom@MBPsomtrMarcus2 A1 % python3 JuliaSet.py
This run took a total of 2.316828966140747 seconds
marcuscarlbom@MBPsomtrMarcus2 A1 % python3 JuliaSet.py
This run took a total of 2.3311450481414795 seconds
marcuscarlbom@MBPsomtrMarcus2 A1 % python3 JuliaSet.py
This run took a total of 2.3112940788269043 seconds
marcuscarlbom@MBPsomtrMarcus2 A1 % python3 JuliaSet.py
This run took a total of 2.308480978012085 seconds
marcuscarlbom@MBPsomtrMarcus2 A1 % python3 JuliaSet.py
This run took a total of 2.3560657501220703 seconds
marcuscarlbom@MBPsomtrMarcus2 A1 %
```

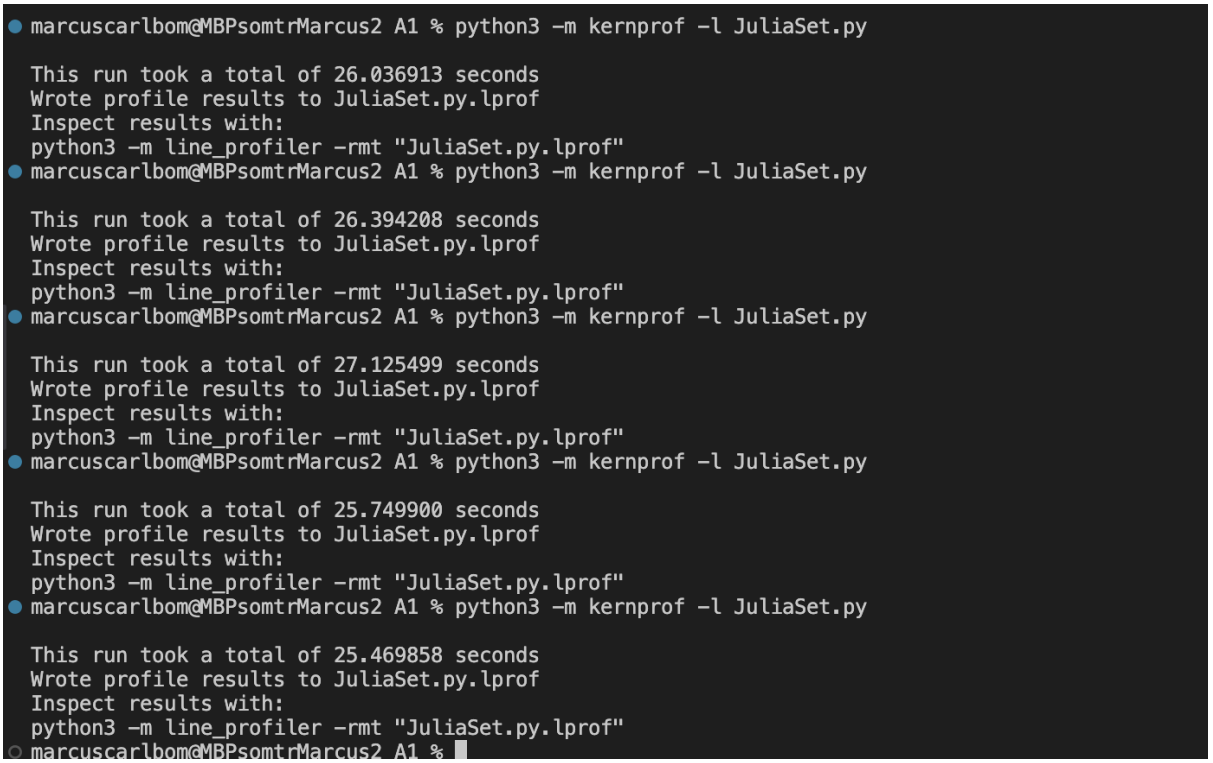
Average time 2.325 seconds

Running JuliaSet.py with cProfiler:

```
marcuscarlbom@MBPsomtrMarcus2 A1 % python3 -m cProfile -o profile.stats JuliaSet.py
This run took a total of 5.087873935699463 seconds
marcuscarlbom@MBPsomtrMarcus2 A1 % python3 -m cProfile -o profile.stats JuliaSet.py
This run took a total of 5.055094003677368 seconds
marcuscarlbom@MBPsomtrMarcus2 A1 % python3 -m cProfile -o profile.stats JuliaSet.py
This run took a total of 5.107017993927002 seconds
marcuscarlbom@MBPsomtrMarcus2 A1 % python3 -m cProfile -o profile.stats JuliaSet.py
This run took a total of 5.09935188293457 seconds
marcuscarlbom@MBPsomtrMarcus2 A1 % python3 -m cProfile -o profile.stats JuliaSet.py
This run took a total of 5.090599298477173 seconds
marcuscarlbom@MBPsomtrMarcus2 A1 %
```

Average time 5.088 seconds, about 2.763 seconds of extra overhead which is about 219% increase in run time.

Running JuliaSet.py with line_profiler:



Average time 26.15 seconds, about 23.825 seconds of extra overhead which is about 11247% increase in run time.

T1.4

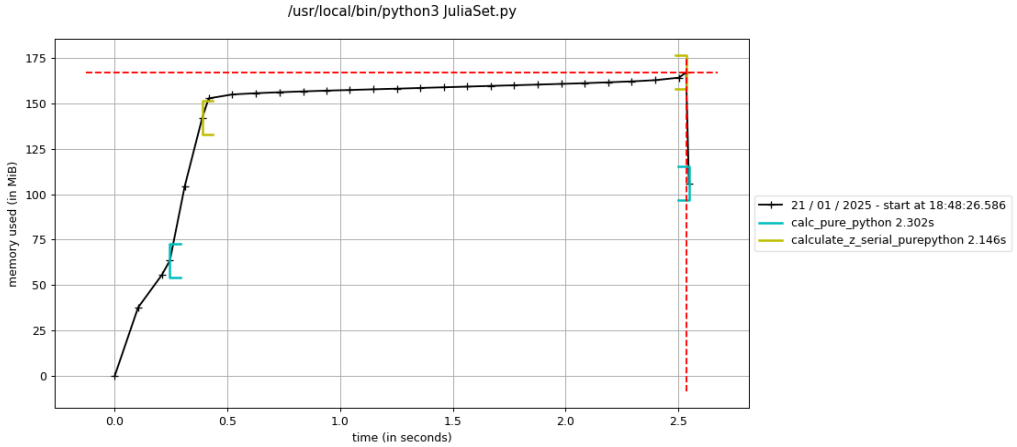
Results from running

```
python3 -m memory_profiler JuliaSet.py
```

```
marcuscarlbom@MBPsomtrMarcus2 A1 % python3 -m memory_profiler JuliaSet.py
This run took a total of 1578.771924 seconds
Filename: JuliaSet.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    28   64.375 MiB   64.375 MiB           1   @profile
    29                                         def calc_pure_python(desired_width, max_iterations):
    30                                             """Create a list of complex coordinates (zs) and complex parameters (cs),
    31                                             build Julia set"""
    32   64.375 MiB    0.000 MiB           1       x_step = (x2 - x1) / desired_width
    33   64.375 MiB    0.000 MiB           1       y_step = (y1 - y2) / desired_width
    34   64.375 MiB    0.000 MiB           1       x = []
    35   64.375 MiB    0.000 MiB           1       y = []
    36   64.375 MiB    0.000 MiB           1       ycoord = y2
    37   64.406 MiB    0.031 MiB        1001       while ycoord > y1:
    38   64.406 MiB    0.000 MiB        1000           y.append(ycoord)
    39   64.406 MiB    0.000 MiB        1000           ycoord += y_step
    40   64.406 MiB    0.000 MiB           1       xcoord = x1
    41   64.438 MiB    0.031 MiB        1001       while xcoord < x2:
    42   64.438 MiB    0.000 MiB        1000           x.append(xcoord)
    43   64.438 MiB    0.000 MiB        1000           xcoord += x_step
    44                                             # build a list of coordinates and the initial condition for each cell.
    45                                             # Note that our initial condition is a constant and could easily be removed,
    46                                             # we use it to simulate a real-world scenario with several inputs to our
    47                                             # function
    48   64.438 MiB    0.000 MiB           1       zs = []
    49   64.438 MiB    0.000 MiB           1       cs = []
    50  140.344 MiB    0.000 MiB        1001       for ycoord in y:
    51  140.344 MiB    0.000 MiB     1001000           for xcoord in x:
    52  140.344 MiB   22.828 MiB     1000000               zs.append(complex(xcoord, ycoord))
    53  140.344 MiB   53.078 MiB     1000000               cs.append(complex(c_real, c_imag))
    54
    55                                             # print("Length of x:", len(x))
    56                                             # print("Total elements:", len(zs))
    57  140.344 MiB    0.000 MiB           1       start_time = time.time()
    58  165.375 MiB  165.375 MiB           1       output = calculate_z_serial_purepython(max_iterations, zs, cs)
    59  165.375 MiB    0.000 MiB           1       end_time = time.time()
    60  165.375 MiB    0.000 MiB           1       secs = end_time - start_time
    61                                             # print(calculate_z_serial_purepython.__name__ + " took", secs, "seconds")
    62
    63                                             # This sum is expected for a 1000^2 grid with 300 iterations
    64                                             # It ensures that our code evolves exactly as we'd intended
    65                                             # assert sum(output) == 33219980


Filename: JuliaSet.py
```

```
Filename: JuliaSet.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    69  140.344 MiB  140.344 MiB           1   @profile
    70                                         def calculate_z_serial_purepython(maxiter, zs, cs):
    71                                             """Calculate output list using Julia update rule"""
    72  147.984 MiB    7.641 MiB           1       output = [0] * len(zs)
    73  165.375 MiB    0.000 MiB     1000001       for i in range(len(zs)):
    74  165.375 MiB    0.000 MiB     1000000           n = 0
    75  165.375 MiB    7.156 MiB     1000000           z = zs[i]
    76  165.375 MiB    7.156 MiB     1000000           c = cs[i]
    77  165.375 MiB    0.000 MiB    34219980           while abs(z) < 2 and n < maxiter:
    78  165.375 MiB    0.000 MiB    33219980               z = z * z + c
    79  165.375 MiB    3.078 MiB    33219980               n += 1
    80  165.375 MiB    0.000 MiB     1000000           output[i] = n
    81  165.375 MiB    0.000 MiB           1       return output


marcuscarlbom@MBPsomtrMarcus2 A1 %
```

mprof plotting



Running the memory profiler 1 time took 1578.77 seconds. This is an increase of 1576.445 seconds which is 679 times longer than just running it.

Running the mprof command

```
python3 -m mprof run JuliaSet.py
```

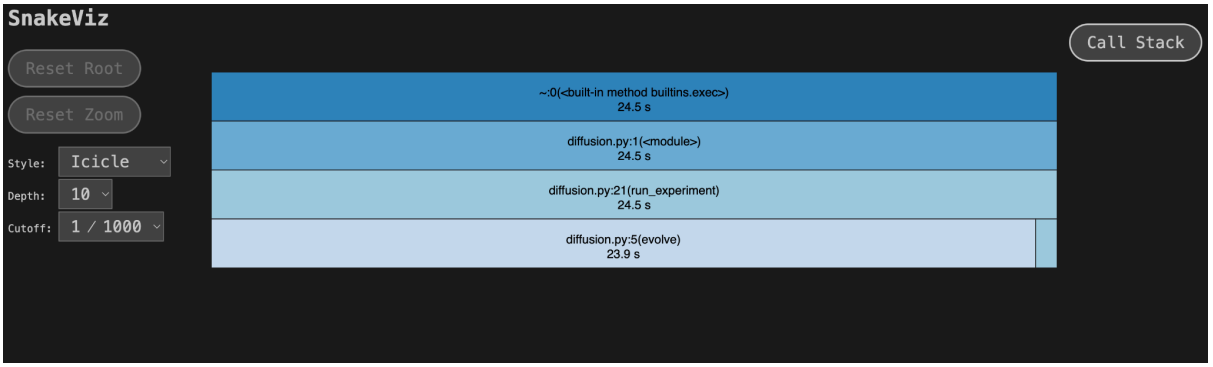
Took on average after 5 runs 2.337 seconds. This is an increase of 0.002 seconds which is about an 0.5% increase in run time. Which due to there being deviations means it is likely having close to negligible impact on run time.

E2

T2.1

```
marcuscarlbom@MBPsomtrMarcus2 E2 % python3 -m cProfile -s cumulative diffusion.py
         304 function calls in 24.634 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   24.634   24.634 {built-in method builtins.exec}
        1    0.003    0.003   24.634   24.634 diffusion.py:1(<module>)
        1    0.631    0.631   24.631   24.631 diffusion.py:20(run_experiment)
      300   24.000    0.080   24.000    0.080 diffusion.py:4(evolve)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}


marcuscarlbom@MBPsomtrMarcus2 E2 %
```

cProfile visualised by SnakeViz



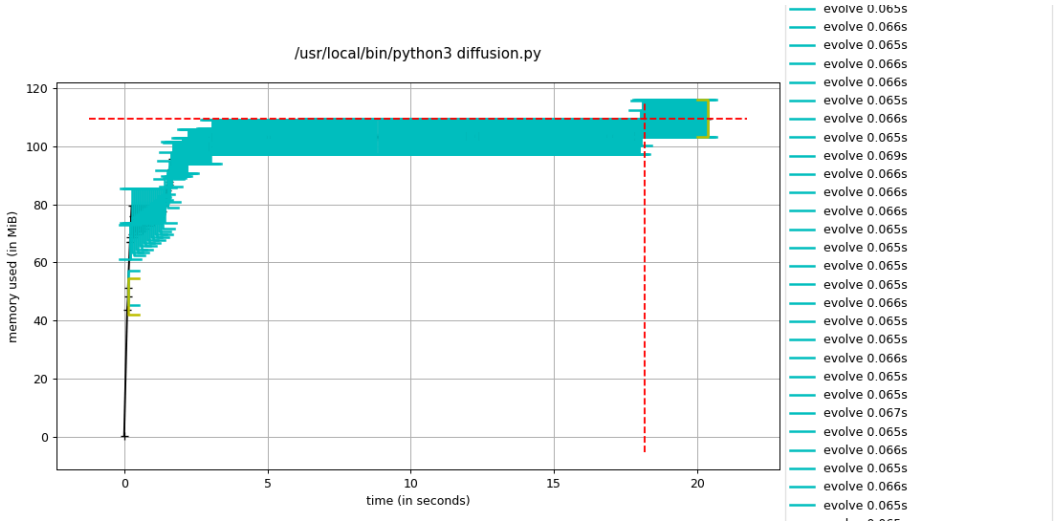
Result of line_profiler:

```
marcuscarlbom@MBPsomtrMarcus2 E2 % python3 -m line_profiler diffusion.py.lprof
Timer unit: 1e-06 s

Total time: 89.5921 s
File: diffusion.py
Function: evolve at line 4

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     4                                           @profile
     5                                           def evolve(grid, dt, D=1.0):
     6       300        184.0      0.6      0.0      xmax, ymax = grid_shape
     7    192300     103921.0      0.5      0.1      new_grid = [[0.0] * ymax for x in range(xmax)]
     8    192300      26586.0      0.1      0.0      for i in range(xmax):
     9 123072000   14349472.0      0.1     16.0          for j in range(ymax):
    10 122880000   10791285.0      0.1     12.0              grid_xx = (
    11 122880000   20288001.0      0.2     22.6                  grid[(i + 1) % xmax][j] + grid[(i - 1) % xmax][j] - 2.0 * grid[i][j]
    12                                                       )
    13 122880000   10590787.0      0.1     11.8              grid_yy = (
    14 122880000   19843522.0      0.2     22.1                  grid[i][(j + 1) % ymax] + grid[i][(j - 1) % ymax] - 2.0 * grid[i][j]
    15                                                       )
    16 122880000   13597930.0      0.1     15.2              new_grid[i][j] = grid[i][j] + D * (grid_xx + grid_yy) * dt
    17       300        369.0      1.2      0.0      return new_grid

Total time: 178.91 s
File: diffusion.py
Function: run_experiment at line 20

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    20                                           @profile
    21                                           def run_experiment(num_iterations):
    22                                               # Setting up initial conditions
    23         1          0.0      0.0      0.0      xmax, ymax = grid_shape
    24       641        812.0      1.3      0.0      grid = [[0.0] * ymax for x in range(xmax)]
    25
    26                                               # These initial conditions are simulating a drop of dye in the middle of our
    27                                               # simulated region
    28         1          1.0      1.0      0.0      block_low = int(grid_shape[0] * 0.4)
    29         1          0.0      0.0      0.0      block_high = int(grid_shape[0] * 0.5)
    30        65          7.0      0.1      0.0      for i in range(block_low, block_high):
    31      4160        467.0      0.1      0.0          for j in range(block_low, block_high):
    32      4096        355.0      0.1      0.0              grid[i][j] = 0.005
    33
    34                                               # Evolve the initial conditions
    35       301        108.0      0.4      0.0      for i in range(num_iterations):
    36       300  178907801.0 596359.3    100.0          grid = evolve(grid, 0.1)
```

Memory_profiler run on diffusion.py

```
marcuscarlbom@Mac E2 % python3 -m memory_profiler diffusion.py
Filename: diffusion.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     4   93.812 MiB 21228.875 MiB         300   @profile
     5                                         def evolve(grid, dt, D=1.0):
     6   93.812 MiB -6734.766 MiB         300       xmax, ymax = grid_shape
     7   93.812 MiB -4309864.156 MiB      192300       new_grid = [[0.0] * ymax for x in range(xmax)]
     8   97.812 MiB -5080139.344 MiB      192300       for i in range(xmax):
     9   97.812 MiB -3252085584.266 MiB   123072000           for j in range(ymax):
    10   97.812 MiB -3247012029.906 MiB   122880000               grid_xx = (
    11   97.812 MiB -3247012162.641 MiB   122880000                   grid[(i + 1) % xmax][j] + grid[(i - 1) % xmax][j] - 2.0 * grid[i][j]
    12                                                           )
    13   97.812 MiB -3247012089.312 MiB   122880000               grid_yy = (
    14   97.812 MiB -3247012077.609 MiB   122880000                   grid[i][(j + 1) % ymax] + grid[i][(j - 1) % ymax] - 2.0 * grid[i][j]
    15                                                           )
    16   97.812 MiB -3247012097.156 MiB   122880000               new_grid[i][j] = grid[i][j] + D * (grid_xx + grid_yy) * dt
    17   97.812 MiB -7887.250 MiB         300       return new_grid


Filename: diffusion.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    20   50.984 MiB   50.984 MiB           1   @profile
    21                                         def run_experiment(num_iterations):
    22                                             # Setting up initial conditions
    23   50.984 MiB    0.000 MiB           1       xmax, ymax = grid_shape
    24   53.953 MiB    2.969 MiB         641       grid = [[0.0] * ymax for x in range(xmax)]
    25
    26                                             # These initial conditions are simulating a drop of dye in the middle of our
    27                                             # simulated region
    28   53.953 MiB    0.000 MiB           1       block_low = int(grid_shape[0] * 0.4)
    29   53.953 MiB    0.000 MiB           1       block_high = int(grid_shape[0] * 0.5)
    30   53.953 MiB    0.000 MiB          65       for i in range(block_low, block_high):
    31   53.953 MiB    0.000 MiB        4160           for j in range(block_low, block_high):
    32   53.953 MiB    0.000 MiB        4096               grid[i][j] = 0.005
    33
    34                                             # Evolve the initial conditions
    35   93.812 MiB -6761.219 MiB         301       for i in range(num_iterations):
    36   93.812 MiB 21242.250 MiB         300           grid = evolve(grid, 0.1)

marcuscarlbom@Mac E2 %
```

mprof plotted



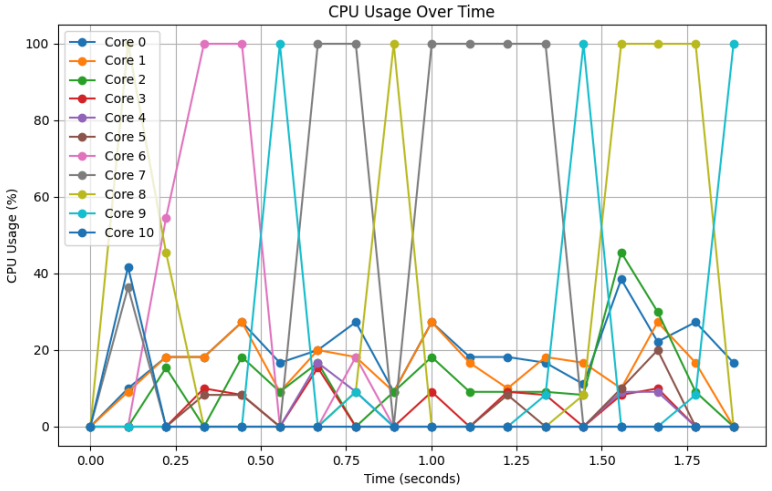
Bonus Exercise:

We made a special class in Python to make it easy to call `with` statement and the wanted interval in seconds. We wanted it to not interfere with the running code so we made a secondary thread using the module “threading” as making that thread sleep was a simple way to handle periodic check of the core usage percentage.

It takes periodic check and saves it to a list called samples. When the given with statement is finished, the __exit__ function ends the thread and makes it possible for calling profiler.plot() and profiler.summary_table() to view the plot of the data and summary table. We did not include minimum values in the summary table as they always had in our tests a value of 0% on each core. The plot() first creates graphs for each respective core and then a combined graph with all of the core percentages over time. We decided to only include the combined graph as the computer used has 11 cores, an unnecessary amount of screenshots.

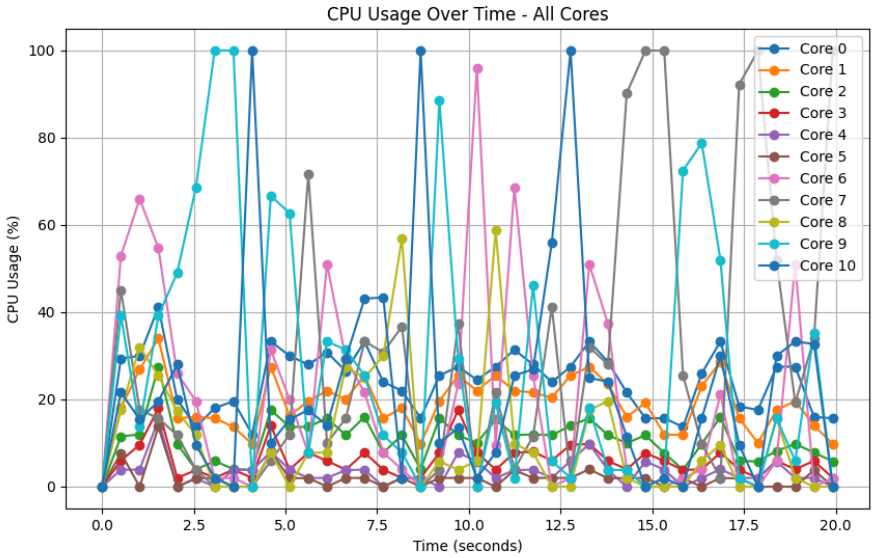
Profiler is seen in the folder A1/B/profiler.py

Result of running JuliaSet code with our own profiler tool at interval 0.1



Core	Avg Usage	Max Usage	Deviation	Time above 50%
Core 0	15.07%	36.40%	8.40%	0.00%
Core 1	10.12%	27.30%	9.19%	0.00%
Core 2	4.43%	10.00%	4.45%	0.00%
Core 3	2.44%	9.10%	3.94%	0.00%
Core 4	1.39%	16.70%	4.17%	0.00%
Core 5	4.54%	54.50%	13.63%	5.56%
Core 6	19.78%	100.00%	31.89%	16.67%
Core 7	18.24%	100.00%	32.06%	16.67%
Core 8	20.53%	100.00%	31.28%	11.11%
Core 9	41.11%	100.00%	43.10%	44.44%
Core 10	10.11%	83.30%	21.16%	5.56%

Result of running diffusion equation code with our own profiler tool at interval 0.5



Core	Avg Usage	Max Usage	Deviation	Time above 50%
Core 0	18.89%	60.00%	15.92%	1.67%
Core 1	14.32%	60.00%	13.19%	0.56%
Core 2	8.61%	50.00%	10.62%	0.00%
Core 3	4.11%	45.50%	7.26%	0.00%
Core 4	1.95%	40.00%	4.73%	0.00%
Core 5	1.43%	46.20%	5.12%	0.00%
Core 6	17.64%	100.00%	24.82%	13.33%
Core 7	28.19%	100.00%	35.33%	21.67%
Core 8	27.79%	100.00%	35.36%	19.44%
Core 9	30.83%	100.00%	36.84%	24.44%
Core 10	11.96%	100.00%	22.96%	6.11%