**1. Line profiler and Numba - Euler’s Method.**

**1.1.** Timer unit: 1e-06 s

Total time: 20.7199 s

File: euler\_ode\_profile.py

Function: euler\_integration at line 25

Line # Hits Time Per Hit % Time Line Contents

==============================================================

25 @profile

26 def euler\_integration(y0, t0, dt, tmax):

27 """

28 This function implements the Euler method to solve

29 an ordinary differential equation given an initial

30 condition (y0) and a constant time increment (dt).

31 The integration is performed until the maximum

32 compute time is reached.

33

34 The function to be integrated is defined in

35

36 int\_funct(y\_i, t\_i )

37

38 INPUTS:

39 - y0: double, Initial value.

40 - t0: double, Initial time.

41 - dt: double, time increment.

42 - tmax: double, Maximum evaluation time.

43

44 OUTPUTS:

45 - y: array float64, Contains all solutions for each time increment

46 - t: array float64, Contains the corresponding time evaluations for each function

47

48 y[i] contains the Euler's solution to the function at time t[i]

49

50 """

51

52

53

54

55 # Compute the number of evaluations.

56 1 1.6 1.6 0.0 nevals = int((tmax-t0)/dt)

57

58 # Initialize arrays.

59 1 9.1 9.1 0.0 y = np.zeros(nevals+1)

60 1 5.0 5.0 0.0 t = np.zeros(nevals+1)

61

62 # Save initial condition.

63 1 1.2 1.2 0.0 y[0] = y0

64 1 0.4 0.4 0.0 t[0] = t0

65

66 # Implement the for loop required to perform Euler's integration

67 #----------

68

69 #----------

70 # Do not modify after this line.

71 10000001 1924309.9 0.2 9.3 for i in range(nevals):

72 10000000 3376508.2 0.3 16.3 t[i+1] = t[i] + dt

73 10000000 15419025.8 1.5 74.4 y[i+1] = y[i] + dt \* int\_funct(y[i], t[i])

74

75 1 4.2 4.2 0.0 return y, t

20.72 seconds - euler\_ode\_profile.py:25 - euler\_integration

**Profiling Results Summary**

Total Execution Time: The euler\_integration function took approximately 20.72 seconds to complete.

**Time Distribution:**

Initialization: The time taken to compute the number of evaluations (nevals) and initialize the y and t arrays is negligible, accounting for an insignificant portion of the total time.

**Loop Execution:**

Updating the time array t[i+1] = t[i] + dt consumes 16.3% of the total execution time.

The bulk of the execution time (74.4%) is spent on calculating y[i+1] = y[i] + dt \* int\_funct(y[i], t[i]).

**Conclusions**

The most time-consuming operation in the **euler\_integration** function is the calculation of the next value of y, which involves calling the **int\_funct** function and performing arithmetic operations. This step alone accounts for nearly three-quarters of the total execution time.

Incrementing the time in the t array also has a significant impact on performance, though to a lesser extent compared to the computation of y[i+1].

Given these insights, optimization efforts should primarily focus on the computation of y[i+1]. Using Numba's JIT compiler (@jit decorator) significantly improves the performance of this operation by compiling the Python code to optimized machine code. This is particularly effective for loops and function calls within loops, which are prevalent in numerical integration tasks.

**1.2.** Solution for time t=10.0000: y(t)=0.83907180

Analytical solution : y(t)=0.83907153 error: 2.7264e-07

CPU time 9.238343238830566

**1.3.** Solution for time t=10.0000: y(t)=0.83907180

Analytical solution : y(t)=0.83907153 error: 2.7264e-07

CPU time 0.8270158767700195

**Performance Results Summary:**

Pure Python Implementation (euler\_ode.py):

Execution Time: 12.551 seconds

This version serves as the baseline for performance comparison.

Single-Function JIT Optimization (euler\_ode\_numba1.py):

Execution Time: 9.238 seconds

Observing a reduction in execution time compared to the pure Python version, this demonstrates the effectiveness of applying JIT compilation to computationally intensive functions.

Full-Function Parallelization with JIT (euler\_ode\_numba2.py):

Execution Time: 0.827 seconds

A dramatic reduction in execution time demonstrates the substantial performance gains achievable through broader application of JIT compilation and parallelization strategies.

**Analysis and Conclusions:**

Impact of JIT Compilation: The transition from a non-optimized Python script to a version with JIT compilation (euler\_ode\_numba1.py) resulted in a noticeable performance improvement. This underscores the power of Numba's JIT compilation in optimizing Python code, especially for numerical computations.

Significance of Full-Function Optimization: Applying JIT compilation to the entire integration function, not just the computational core (int\_funct), as done in euler\_ode\_numba2.py, led to an even more significant performance gain. This version showed an impressive reduction in execution time, which highlights the potential of comprehensive optimization strategies, including leveraging parallel execution capabilities where applicable.

Efficiency of Parallelization: The most considerable performance enhancement was observed with the application of Numba’s JIT compilation across the full function (euler\_ode\_numba2.py), which suggests that the parallel execution capabilities of Numba (enabled by the parallel=True flag) were effectively utilized. This suggests that the iterative calculations in the Euler integration method are well-suited to parallelization, leading to superior speed improvements.

**2. Numba – Automatic parallelization.**

Starting job - Tue Apr 9 01:10:36 CDT 2024

Running with NUMBA\_NUM\_THREADS=1

Integral 1.804776

CPU time:2.668961s

Calculations ended with NUMBA\_NUM\_THREADS=1

Total CPU time 5438 [ms]

----------------------------------

Running with NUMBA\_NUM\_THREADS=2

Integral 1.804776

CPU time:2.048761s

Calculations ended with NUMBA\_NUM\_THREADS=2

Total CPU time 2672 [ms]

----------------------------------

Running with NUMBA\_NUM\_THREADS=4

Integral 1.804776

CPU time:2.116773s

Calculations ended with NUMBA\_NUM\_THREADS=4

Total CPU time 2726 [ms]

----------------------------------

Running with NUMBA\_NUM\_THREADS=8

Integral 1.804776

CPU time:2.114511s

Calculations ended with NUMBA\_NUM\_THREADS=8

Total CPU time 2720 [ms]

----------------------------------

Running with NUMBA\_NUM\_THREADS=16

Integral 1.804776

CPU time:2.117159s

Calculations ended with NUMBA\_NUM\_THREADS=16

Total CPU time 2676 [ms]

----------------------------------

Running with NUMBA\_NUM\_THREADS=20

Integral 1.804776

CPU time:2.113811s

Calculations ended with NUMBA\_NUM\_THREADS=20

Total CPU time 2725 [ms]

|  |  |  |  |
| --- | --- | --- | --- |
| **Number of Processes** | **CPU time[s]** | **Speedup** | **Efficiency** |
| **1** | 2.668961 | 1 | 1 |
| **2** | 2.048761 | 1.30271955 | 0.651359773 |
| **4** | 2.116773 | 1.26086312 | 0.315215779 |
| **8** | 2.114511 | 1.26221193 | 0.157776491 |
| **16** | 2.117159 | 1.26063324 | 0.078789577 |
| **20** | 2.113811 | 1.26262991 | 0.063131496 |
|  |  |  |  |
| **Serial time** | 2.668961 |  |  |

**3. Bonus: Cython – Matrix-matrix multiplication.**

Cython Implementation:

Size: 3x3, Avg Time: 0.000001 seconds

Size: 10x10, Avg Time: 0.000002 seconds

Size: 100x100, Avg Time: 0.001016 seconds

Size: 1000x1000, Avg Time: 1.209425 seconds

NumPy dot Function:

Size: 3x3, Avg Time: 0.000769 seconds

Size: 10x10, Avg Time: 0.000009 seconds

Size: 100x100, Avg Time: 0.000248 seconds

Size: 1000x1000, Avg Time: 0.003263 seconds