# Part 1

Overall, a raw Python formulation runs fast, only taking ~23.5ms to perform the Euler’s method initial value problem, shown in Table 1. The error shows as zero, as the error is smaller than the precision the table is displaying at.

Table 1. Performance of Euler's Method IVP Solve.

|  |  |  |
| --- | --- | --- |
| **Run** | **Runtime [s]** | **Error** |
| *Raw Python* | 10.2E+0 | 0.000% |
| Numba-1 | 6.92E+0 | 0.000% |
| Numba-2 | 1.23E+0 | 0.000% |

To improve the performance of the Python code, we can utilize Line Profiler to understand which functions and where are taking the most time to run. The output of this utility is shown in Figure 1. Clearly overall, the calling of the function that accounts for the most time in producing the integral time. Due to being called 10,000 times, the function takes the most time to add its data to the calculation. The total time of this call is ~45ms, which is an order of magnitude more than the next longest calculation.

The simplest way to improve this calculation is to find a way to do all these calculations in parallel, which would preferably done via Numba, which can accelerate these functions.

The “jit” decorator can be added to each of the functions, allowing Numba to compile these functions outside of Python. However, for the large calculation it may be beneficial to allow Numba to compile the calculation for a small set and then run for the larger set. If the “jit” decorator is added to only the function that contains the differential equation to be solved, the process is sped up some, seen in Table 1. Ultimately, this is not the bulk of the speedup available as there is not a recurring activity within this function.

The “jit” decorator can also be added to the function that performs the calculation to allow for that calculation to be done in parallel. Seen in Table 1, this is where a bulk of the speedup comes from because that recurring “for” loop is now no longer in the Python environment, but compiled by Numba.

One could also note that I set a separate allocation call for the gradient going into Euler’s method. This is because the code required this allocation for improved speedups. Numba seems to need a pre-allocation of as much as possible to optimize its performance.

Overall, the best way to utilize Numba appears to be moving as many recurring calculations as possible outside of the Python interpreter.

A computer screen shot of a black screen

Description automatically generated

Figure 1. Line Profiler Output for Euler’s Method IVP.

# Part 2

We can further utilize Numba by allowing it to parallelize calculations in its complation. Instead of doing the Euler’s method, a numerical integration is set up in this section and Numba’s “jit” decorator has a parallel condition that is supported by the prange method added to the conditions of the “for” loops. The script was run multiple times with varying numbers of Numba threads, shown in Table 2.

Table 2. Numba Thread Performance Improvement.

|  |  |  |  |
| --- | --- | --- | --- |
| **Numba Threads** | **Runtime [s]** | **Speedup** | **Efficiency** |
| 1 | 2.88 | 1.00 | 100% |
| 2 | 1.74 | 1.65 | 82.6% |
| 4 | 1.60 | 1.80 | 45.0% |
| 8 | 1.27 | 2.28 | 28.4% |
| 16 | 1.03 | 2.81 | 17.5% |
| 20 | 0.980 | 2.94 | 14.7% |

When this data is plotted, there is a clear improvement in performance with more threads, but a diminishing return in the efficiency, seen in Figure 2. Likely with larger calculations, a continued improvement in performance would have been seen for increased Numba threads. However, with larger machines it is impractical to only need a few or a single Numba threads.

Figure 2.

## Part 3

Another way perform various functions is to utilize Cython. In this case, a simple matrix multiplication is formulated in Cython.

Overall, Cython performs well for small matrices, as seen in Table 3. However, the Cython function does not perform well for larger matrices, where the NumPy architecture likely dominates the performance. This happens around a matrix size of 70, according to Figure 3. This indicates that the Cython pre-compilation offers excellent performance benefits for small calculations, but may lag behind larger calculations with better architectures.

Table 3. Performance Benchmarks of NumPy and Cython Matrix Multiplication.

|  |  |  |  |
| --- | --- | --- | --- |
| **Matrix Size** |  | **Runtime [s]** | |
|  | **NumPy** | **Cython** |
| *3* |  | 6.99E-3 | 10.8E-6 |
| 10 |  | 1.90E-3 | 19.9E-6 |
| 100 |  | 3.94E-3 | 9.81E-3 |
| 1000 |  | 10.2E-3 | 4.17E+0 |

Figure 3.