Trapezoidal Rule - Comparing Python, NumPy and Cython

Keshav Saravanan, EE23B035

October 2024

In this assignment, I write an optimized Cython implementation of the trapezoidal integration rule.

1 Background Information

Python is an interpreted language, and is extremely slow - however it can be sped up with bindings written in C through Cython. There are multiple steps that can be optimized in this problem, however it is not as simple as translating the Python implementation into Cython bindings - there are overheads involved in calling a Cython function and it must be ensured that these overheads do not negate the performance gains from C.

2 Running the Project

All of my code is written in a Jupyter Notebook with multiple benchmarks - they can be run by running each cell sequentially. However, if you wish to rerun a benchmark, make sure to rerun all the cells above it that are under the same section, because the `FUNCTIONS` and `IMPLEMENTATIONS` variables are overwritten for each benchmark.

3 Approaches

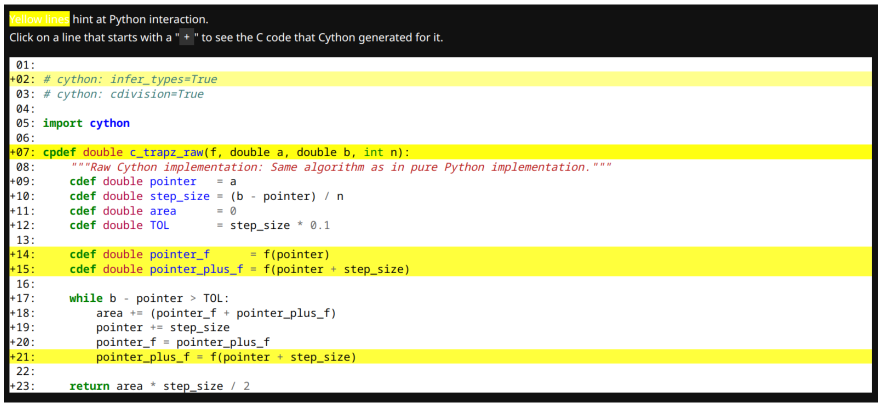
I optimized my Cython implementation in iterations - applying optimizations in stages, and benchmarking them to measure the performance gains.

3.1 Python Implementation of Math Functions

For the base standard, I implement both the Python and Cython versions the same way.

- In both the Python and Cython implementations, I loop from the lower limit (`A`) to the upper limit (`B`) of the integration. The step size is $\frac{B-A}{N}$, where `N` is the number of trapezoids to use.
- The Cython implementation turns on `cdivision` - so Cython does not check for zero division while performing a division. This speedup is not noticeable at all at this stage, since the only division being performed is to find the step size at the start of the function call.
- The math functions are directly provided from the `math` library - they are written in Python.
- Because of this, the NumPy implementation must vectorise the input function to allow it to operate on a NumPy array. It uses `linspace` to generate an x-array, and vectorises the input math function to obtain a y-array. It uses the in-built `trapz()` method to calculate the area.

The benchmarking showed that the Cython implementation was much faster than the NumPy and Python implementations, as expected. However, there were still quite a few calls to the Python interpreter (in yellow) that was slowing down the Cython implementation.


Yellow lines hint at Python interaction.
Click on a line that starts with a " + " to see the C code that Cython generated for it.

```

# cython: infer_types=True
# cython: cdivision=True

import cython

cpdef double c_trapz_raw(f, double a, double b, int n):
    """Raw Cython implementation: Same algorithm as in pure Python implementation."""
    cdef double pointer   = a
    cdef double step_size = (b - pointer) / n
    cdef double area      = 0
    cdef double TOL       = step_size * 0.1

    cdef double pointer_f      = f(pointer)
    cdef double pointer_plus_f = f(pointer + step_size)

    while b - pointer > TOL:
        area += (pointer_f + pointer_plus_f)
        pointer += step_size
        pointer_f = pointer_plus_f
        pointer_plus_f = f(pointer + step_size)

    return area * step_size / 2
```

Figure 1: Baseline C Implementation

3.2 NumPy Implementation of Math Functions

Now, I implement the math functions using NumPy's math library. This causes a significant increase in speed to the NumPy implementation because I no longer need to vectorise the input function, it can be called directly on the NumPy array with no calls to the Python interpreter.

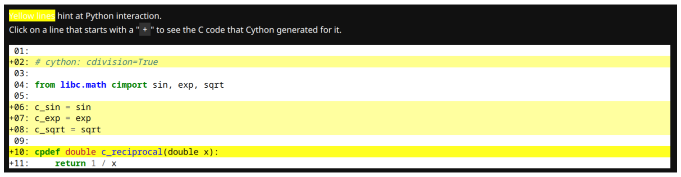
The implementations for the Python and Cython versions stay the same because NumPy's math functions are capable of operating on scalars. However, their speed significantly decreases - this is probably due to the overhead involved in invoking the Cython backend for each and every trapezoid in the integration range.

Noteably, the NumPy implementation from this benchmark was significantly faster than the Cython implementation from the previous benchmark.

3.3 Implementating Specific Math Functions

In this benchmark, I provide each implementation (Python, Cython and NumPy) a math function that is optimized to run fast with that implementation. This means that the Python integrator got the standard math library functions (no overhead in invoking Cython), the Cython integrator got the C implementations from `libc.math` and the NumPy implementation got the `numpy.math` functions.

In this case, enabling `cdivision` helps a lot in the case of `f(x) = 1/x`: If we know that `x` is never 0, we can optimize those checks away.


Yellow lines hint at Python interaction.
Click on a line that starts with a " + " to see the C code that Cython generated for it.

```

# cython: cdivision=True

from libc.math cimport sin, exp, sqrt

c_sin = sin
c_exp = exp
c_sqrt = sqrt

cpdef double c_reciprocal(double x):
    return 1 / x
```

Figure 2: C Implementations of the Math Functions

In this benchmark, the Cython implementation performs nearly as fast as the NumPy version.

3.4 Optimizing the Cython Implemented Math Functions

Even though my Cython implementation is now of comparable performance as the NumPy implementation, I believed there were more optimizations that could be performed. Even though the C implementations of the math functions, like `sqrt()` and `exp()` were in C, my integrator function did not know that, and was treating it as a Python `object`. This can be seen in the fact that those function calls were highlighted in yellow in the output annotations. This meant that every time the function was called, it was handed off to the Python interpreter - far from ideal.

It is not possible to directly pass a function as a parameter into Cython - it will not be able to handle the Python objects it cannot infer anything about. So while the function is written in C, there is a lot of overhead in calling it - Cython must pass this off to the Python interpreter, which then invokes the respective Cython binding once again.

With the given function format, it is not possible to fix this. However, I deviated slightly from this format to allow the first parameter to be a Cython class (with it's `evaluate()` method being the input function to integrate. That way, there's not overhead involved in calling the Python interpreter. This is seen in the fact that the annotations now contain almost no yellow highlighting. The benchmarks for this were quite surprising - the Cython implementation was 2-3 times as fast as the NumPy implementation.

3.5 Implementing Parallelism

Now that the Cython implementation does not require any calls to the Python Interpreter, it could theoretically be parallelized, since it would not be limited by the GIL. I tried this out, with disappointing results - the speed of the Cython implementation actually reduced a little, probably due to the overhead in spawning so many threads.

4 Results

In the end, my Cython implementation ended up being much faster than the NumPy implementation - upto 5 times as fast in the best case. When benchmarking on $f(x) = x^2$, from 0 to 10, the Cython implementation was similarly faster (about 8x the speed of the NumPy implementation).

My Cython implementation works with the `double` datatype, meaning it is less precise than NumPy's `float64` implementation (which is the default). However, the results were still accurate to multiple significant digits. Changing the Cython implementation's datatype to a `float` did not result in any noticeable speed gains.

I think the reason my Cython implementation is much faster than NumPy's implementation is because my code is heavily optimised for the problem at hand. NumPy's functions, on the other hand, are meant to be more generic and are written for ease of use from Python, so there would be a lot of overhead involved in using them (related to memory handling, calling Python objects, making no assumptions about the datatype and shape of the input array, etc).

5 Sources

I relied mostly on the documentation to write my Cython code -

1. Extension Types - https://docs.cython.org/en/latest/src/tutorial/cdef_classes.html
2. The GIL - https://cython.readthedocs.io/en/latest/src/userguide/nogil.html
3. Parallelism in Cython - https://cython.readthedocs.io/en/latest/src/userguide/parallelism.html