Optimizing with Cython

Krutarth Patel

October 27, 2024

1 Introduction

This report presents a comparision of speed and accuracy between vanilla Python, Cython and numpy and C. And to that end I implement an integral evaluating function using all four.

2 Usage

I have included two shell scripts to display all the results shown in this report. This script would work on any UNIX device given you have gcc and the usual python python libraries installed(ex: matplotlib, numpy, cython)

1. Use `run.sh` personal computer
2. Use `run_jupy.sh` to get results on the jupyter server. Timings and answers for all tests are generated in `data.txt`. Please use this when running on the jupyter server since plotting is not possible.

following section is for Windows users: please execute the following commands(they are written in the shell script as well).

```
python3 setup.py build_ext --inplace
python3 -c "import cy_mod" > data.txt
python3 py_mod.py >> data.txt
python3 plot.py
```

You can optionally run the C program by compiling `trapz.c` using a compiler of choice. for gcc, do:

```
gcc trapz.c -lm
```

NOTE: please pipe the output of the program using `>>` to `data.txt` like so:

```
a >> data.txt
```

This is required since `plot.py` reads from this file to plot all the implementations.

3 Implementation Details

3.1 Cython

typing even some of the variables as ctypes improves the speed by 2x:

```
1
2  def cy_trapz(f: Callable[[float], float],
3        a: cython.double,
4        b: cython.double,
5        n: cython.int) -> cython.double:
6      delta  = (b-a)/n # Python Object and not ctypes
7      prev_y = f(a)
8      acc : cython.double= 0.0
9      x: cython.double = a + delta
10     for i in range(1,n+1):
11         curr_y: cython.double = f(x)
12         acc += ((curr_y + prev_y)/2)*delta
13         prev_y = curr_y
14         x += delta
15     return acc
```

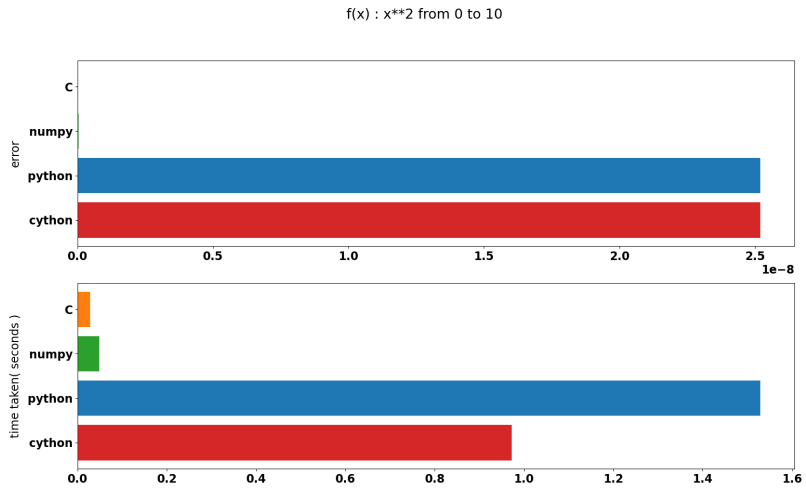


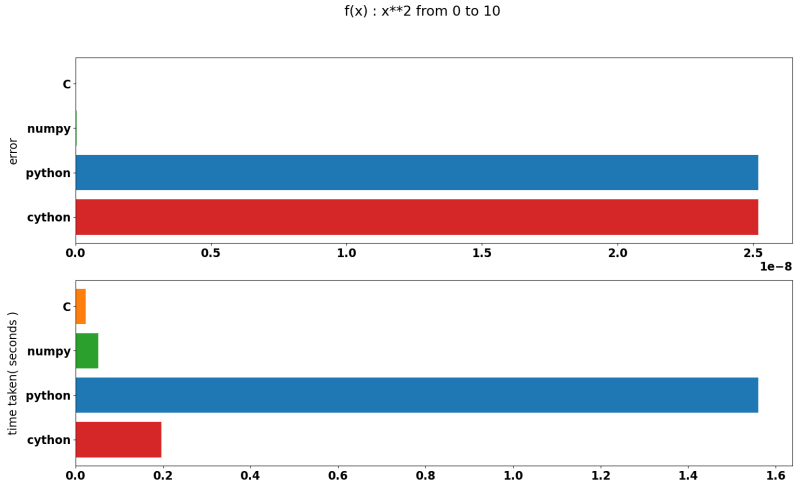
Figure 1: basic implmentation

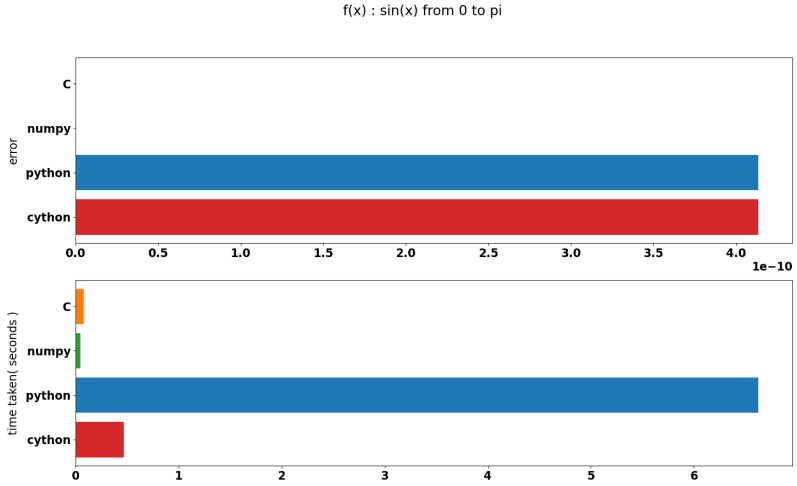
But, we can do better. Explicitly specifying all the types, improves the speed by a significant amount:

```
1  def cy_trapz(f: Callable[[float], float],
2        a: cython.double,
3        b: cython.double,
4        n: cython.int) -> cython.double:
5      delta: cython.double = (b-a)/n
6      prev_y: cython.double = f(a)
7      acc : cython.double = 0.0
8      x: cython.double = a + delta
9      for i in range(1,n+1):
10         curr_y: cython.double = f(x)
11         acc += ((curr_y + prev_y)/2)*delta
```

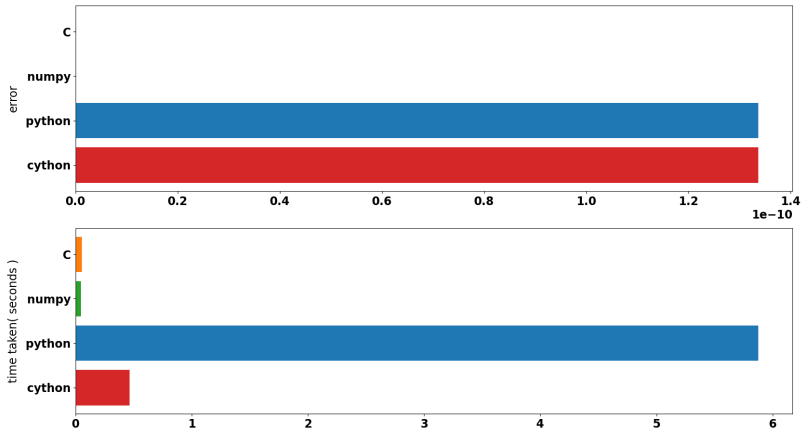
```
12          prev_y = curr_y
13          x += delta
14      return acc
```

Finally, here are all the test results:

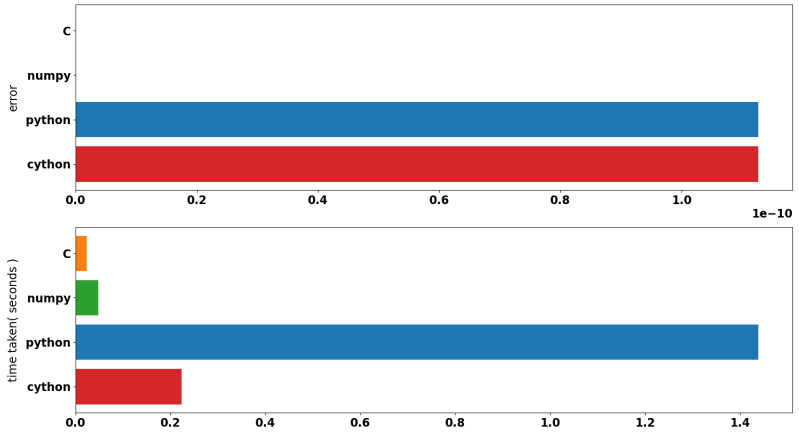




f(x) : ex from 0 to 1



f(x) : 1/x from 1 to 2



4 Observations

1. Python implementation is the slowest and least accurate. One reason could be poor memory management by the interpreter although that is not evident in this example since we are not using sequence types, but is a fact.

2. Cython implementation is almost as fast as numpy. But not quite, because of the bindings between python API and C which add some time. Notably, conversion of objects from Python Objects to c-types is one factor.

3. Numpy implementation is really fast. It beats C in one of the tests! I found the reason to be effecient memory management. Since `np.trapz()` takes an `np.array` as argument which is known to be order of magnitudes faster than python `list` due to more cache hits.

4. C implementation beats all of them except numpy in one test case, notably `sin()` implementation in numpy seems to be faster than its C counterpart in `math.h` .