Optimizing and writing docstrings for TOFu analysis code

Benjamin Eriksson

Uppsala University

February 26, 2021

1 INTRODUCTION

In this project I have worked on the TOFu data analysis code I have written. The code analyzes up to a few gigabytes of data acquired by the TOFu acquisition system for a given experiment and produces a time-of-flight neutron spectrum by finding coincidences between data from different detectors. It also plots the neutron energy spectrum. The code performs, among other things, a coincidence analysis on time-stamp trains and calculates the integral of the pulse waveforms.

For large data sets the code takes a long time to run. The main goal of this project is to find where the code can be optimized by using `line_profiler` and to implement optimizations at these locations in the code. The code currently already runs on 16 cores simultaneously on a computer cluster, thus, the optimization will be focused on implementing faster numpy/scipy algorithms or by Cythonizing the problematic parts of the code. The secondary goal is start writing docstrings for the functions in the analysis code and to use Sphinx to generate a web page for the docstrings.

Some comments

1. The code will not run on a local computer as it downloads data from a given experiment off the servers provided by the experimental facility
2. `create_TOF.py` is the main script used to run the analysis.
3. `functions/tofu_functions.py` contains all the functions used by the main script.
4. The optimization work is performed in `optimization/`

2 OPTIMIZING THE CODE

From previous attempts of optimizing the code I have included the possibility to time each function in `tofu_functions.py` by setting an input argument in the main script. The output this generates for one of the 16 processes is shown in `optimization/basic_timing.txt`. The single longest time spent on a function is related to downloading data from a server. As there is nothing I can do to optimize this function I ignore it. Three other functions were identified as time-heavy, namely

1. `sTOF4()` - finds coincidences between time stamps
2. `get_pulse_area()` - integrates all the pulses
3. `time_pickoff_CFD()` - finds the time-of-arrival for all pulses

As a second step I ran `line_profiler` on these three functions to figure out where in each function time was spent. There were issues with running the `line_profiler` while using multiprocessing, so instead I tested the three functions using some locally stored data for a single process in sub-directories of `optimization/`.

sTOF4()

Running `line_profiler` on the `sTOF4()` function yielded the result shown in `optimization/sTOF/sTOF4.txt`. 25% of the time is spent on the `np.searchsorted()` function. One immediate fix was to move `np.searchsorted()` outside the for-loop to perform the search once using the full array of `w_low` instead of doing it for each element in `w_low`. This increased the overall speed of the function by a factor 2. I then Cythonized the function in `optimization/sTOF/CyTOF.pyx`. This yielded an additional speed increase by a factor 16 as can be seen in `optimization/sTOF/CyTOF.txt`. Thus, the total speed-up after performing these two actions was approximately a factor 32.

get_pulse_area()

As can be seen in `optimization/get_pulse_area/gpa.txt` all of the time is spent on the function `np.trapz()` which calculates the area under the pulses. Cythonizing this function would not yield any significant increase in speed.

time_pickoff_CFD()

As seen in `optimization/time_pickoff_CFD/tp_CFD.txt` almost half the time of the `time_pickoff_CFD()` function is spent on performing the calculations in the `find_points()` function. In turn, `find_points()` spends most of its time subtracting a 1D array from a 2D array. I rewrote the function using Cython to test if it would be quicker to perform the subtraction in a loop instead of using numpy. This slowed the function down by a factor 6. Cythonizing and keeping the numpy calculation made no significant difference in speed as can be seen in the line profiler output `optimization/time_pickoff_CFD/Cy_CFD.txt`. In the end I decided to keep the original function.

3 WRITING DOCSTRINGS

I have written docstrings for X of the functions in `functions/tofu_functions.py`. Using Sphinx a HTML file is generated. It can be found under `docs/_build/html/index.html`. Most of the functions only contain a comment. For this project I have rewritten some of these properly, and the plan is to continuously improve the docstrings.

4 CONCLUSIONS

The TOFu analysis code has been analysed using a basic function timer to find which functions are most time-demanding. Three functions were identified as especially time-demanding. These were analysed using `line_profiler` and rewritten by implementing quicker numpy functions or by cythonizing the function. In the case of the function `sTOF4()`, a speed increase by a factor 32 was achieved by cythonizing the function and better applying a numpy function to the calculation. An attempt to increase the speed of the two other functions, `get_pulse_area()` and `time_pickoff_CFD()`, was performed, however these are likely already written in a speed-efficient way using the numpy library. Attempts to cythonize these slowed them down.

Docstrings for 8 functions have been written and an HTML file generated using Sphinx. Docstrings for the remaining functions will be rewritten properly outside the scope of this project.