Cythonizing Python Code **KAM 5/11/2020**

Summary.

Python is immensely suited for scientific programming where grouped data needs to be passed from function to function, and we are using it plus the NumPy package for simulating Wright Group spectra. The NumPy package is extremely useful for generating multi-dimensional arrays used in CMDS. However, preservation of the structure of the arrays and grouped data surrounding them may slow down intensive computations when they are involved. As a means of checking the efficiency of code and for pedagogical reasons, I studied the use of Cython as a means of speeding up bottlenecked code wherein the Numpy-based computation was reduced to C arithmetic.

I was particularly interested in the matrix propagation section (Kohler et al JCP article) of current CMDS simulations as I reasoned that this multi-element computation section was the "slow part" of CMDS code. This was addressed in great detail in Sunden et al.'s SciPy 2018 Proceedings paper.

The following studies were made:

1. Ratio of "Cythonized" code vs Python code speed for a modified script in WrightSim called "target.py", using a modified sub-script of "propagate.py" found in the WrightSim>>Hamiltonian >> mixed folder. Cythonized code details are available, but examination of the cython file could prove sufficient for interested parties.
2. Similar studies with Cython's (current) OpenMP parallel computing support, as the matrix-based, multi-element computation should prove easily parallelized.

Study 1. Cythonized variant of propagate.py

The procedure is documented in the group's Google Drive. Briefly, a .pyx file is created to simulate the for loop in propagate.py. This pyx file is mainly Python-like in appearance and execution, and it utilizes the notion that the Hamiltonian matrix described in S15 in Kohler et al is a static 9 x9 with multiple zero-entry elements. The matrix is 23/81 = 28% populated which is sufficient for an element by element multiplication consideration but not considered sparse enough for sparse matrix manipulation methods. The pyx file was converted to .c code via Cython then compiled to a shared object file ('.pyd' or .so for Linux). This shared object is imported like a Python module. Further details are described in the procedure. Coding was straightforward and can be analyzed as per user.

A timing function was used to compare execution times between the Cythonized and non-Cythonized variants of propagate.py. The timing function starts prior to the call to exp.run and ends after execution, and so the timing includes script functions outside of the Cythonized portion of the propagation step.

Parameters used in target.py:

```
dt = 50.  # pulse duration (fs)
slitwidth = 120.  # mono resolution (wn)
nw = 16  # number of frequency points (w1 and w2)
nt = 20 # number of delay points (d2)
```

```
exp = experiment.builtin('trive')
    exp.w1.points = np.linspace(-
2.5, 2.5, nw) * 4 * np.log(2) / dt * 1 / (2 * np.pi * 3e-5)
    exp.w2.points = np.linspace(-
2.5, 2.5, nw) * 4 * np.log(2) / dt * 1 / (2 * np.pi * 3e-5)

    exp.d2.points = np.linspace(-2 * dt, 8 * dt, nt)
    exp.w1.active = exp.w2.active = exp.d2.active = True

    exp.timestep = 2.
    exp.early_buffer = 100.0
    exp.late_buffer  = 400.0
```

Results and Discussion. The script was performed with two different modes. The first mode leaves the multiprocessing flag blank ('False') as in the later target.py line:

```
scan = exp.run(ham, mp='')
```

The other mode ran with mp='cpu'. The CPU is an Intel i7-8700K(6 core/12 thread) CPU @ 3.7 GHz with modest overclocking. Time results were averaged over 10 runs without regard for minimization of other activities on the computer. Due to the methods of timing, the y-intercept of a time vs. 'nt' parameter will be essentially zero, and cursory investigations indicated that time scaled precisely linearly with 'nt'. Standard deviations were not incorporated. Results are tabulated.

Mode	Non-Cythonized	Cythonized
mp = ' '	14.73 sec	1.53 s
mp= 'cpu'	4.27 s	2.13 s

Some observations were made with timings through alternate pyx files referencing internal Numpy arrays as opposed to C arrays. There, code executed orders of magnitude slower. It would be useful for experimenters to see how code speed changes if numpy arrays are used internally. Internal memory allocation using `malloc` was not considered but may be worth investigation. The thought was that the viewing of the larger 3D array in the heap was one of the primary bottlenecks and that using stack memory for smaller arrays would be inconsequential in speed increase.

An extremely thorough search of numpy and CBLAS libraries was made. At no instance in the normal matrix ("dot") -vector product libraries was there a provision for discounting a multiplication step if the matrix element was zero. The CBLAS libraries have provisions for triangular matrix multiplication...however, these provisions were not integrated into the code for normal matrix-vector multiplication and so it is unavailable in that scope. Brief studies with the low-level libraries indicated that the triangular dot product did not speed up code, and it is likely that the Numpy wrapping is more responsible for code slowdown than the actual multiplication steps. Investigations into sparse matrix methods were only brief as the 9 x9 matrix is not deemed large and sparse enough for speed improvements with these methods.

Study 1 Conclusions. There is no multiprocessing boost with the Cythonized code as it currently stands as this C code is unable to take advantage of multicore steps. The speed with mp set to 'cpu' is indeed slower. Slowdown may be related to global interpreter lock calls that are not useful here due to this code being optimized for single-threaded operation.

On the other hand, the multiprocessor output does not reduce computation time of the non-Cythonized code by a factor of 1/number of cores (=1/6 here). The reduction is closer to 1/4. Further, the Cythonized code time without the multiprocessor capability is still ~ 1/3 that of the non-Cythonized code with multiprocessor capability.

It is believed that the bulk of the 0.076 second time is not in the propagation portion of the code but in other portions leading to the code. Indeed, timing functions indicate the time of the Cythonized code is approximately 6 x 10-5 sec with nearby instantiating of the arrays at 1.6 x 10^-4 sec. The latter number is in keeping with a typical near 0.5 MB array instantiation as is the case with the 3-dimensional Hamiltonian. The former is roughly 3-4 times the expected number of CPU cycles expected in the cythonized code assuming a multiplication/addition to be a single clock cycle and array indexing of similar order. The likeliest bulk of the extra time is from the instantiation of a numpy array and copy pasting of values into the array to be passed out into the Python code. From an optimization standpoint, therefore, it is believed unlikely to refine speed faster than the ~0.076 sec time unless other portions of the code (*viz*., outside of the propagation script portion) were improved, with a total speed of a few times the summed 2.0 x 10^-4 sec above being the ultimate target.

The Hamiltonian matrix used in propagation consists of m x m elements. The current code is based on matrix-vector multiplication as the bottleneck step. For a complex matrix-vector multiplication, the number of arithmetic steps in this multiplication, assuming the real and imaginary components are separated at beginning and end, is equal to (m x 4) + (m x 2) where the first term represents the number of multiplication steps and the second the addition steps. The propagation code, based on 2nd-order Runge-Kutta methods, contains 2 of these matrix-vector multiplication steps.

The Numpy dot matrix multiplication step (as of v1.19) does not contain speed provisions for triangular matrix-vector multiplication nor does it contain provisions for specific elements set to zero (though CBLAS packages can be utilized for Numpy builds with at least a triangular multiplication build). Thus, the Cythonized code was set to process only the 23 non-zero elements of relevance of the m=9 matrix in equation S15 (yielding 138 total steps of multiplication and addition compared to 486 for a full 9x9 matrix multiplication). This should in principle reduce computation relative to a standard matrix vector multiplication by a factor of 23/81 = 0.284. That the actual results showed an improvement of nearly 0.103 indicates the Numpy array dot product method contains additional code that slows its computation. Where this slowing occurs is not clear but may have to do with the referencing and dereferencing of Numpy arrays during many internal operations.

Study 2. CPU Parallelization of Cythonized Code (OpenMP).

Currently, the Cython code does not support parallelization. However, element by element inspection indicates that a parallelized version may be possible. Due to the requirement that matrix elements must be multiplied first then summed, A 23 non-zero element matrix-vector multiplication step could be parallelized up to a factor of 23 x 4 =92 which is well under the number of modern GPU CODA cores but strongly above the number of typical CPU cores. Further, extension of the propagator method into Hamiltonians of larger size and/or more non-zero elements supports the notion of a CUDA method for propagating the Hamiltonian.

A strictly CPU-parallelizable method could be established where the individual density elements define the CPU threads. However, note that some of the rows of the Hamiltonian have more non-zero elements. Density elements of that row will take longer to compute with this method, and so this parallelization approach is non-ideal. An ideal one would require the total number of CPU cores or threads and divide up the primary C based for loop (see code) by a second, nested loop of steps similar to modern day core counts. This division of code, while practical, seems obviated by CUDA methods, and it may also be made less practical when considering the global lock polling requirements needed for Python (3.7) parallelization methods. Global lock release would be required within the OpenMP portion of any Cython code created. Reacquiring the lock before return may take a substantial amount of time...more so than the code itself. Thus, this study is only discussed and not implemented in any actual code.

Extrapolation: CUDA parallelization.

Each CUDA core would ideally perform one matrix element multiplication step which is immediately intuitive and practicable. Cython does not have CUDA support. In order to make the best use of CUDA it is currently envisioned to use Compyle (https://compyle.readthedocs.io/en/latest/), numba (https://devblogs.nvidia.com/numba-python-cuda-acceleration/), or a continuation of Sunden et al. Compyle has characteristics similar to Cython and it may be possible in the near future to use it as an alternative method for compiling Python/ hybrid code for CUDA. Numba appears to be the go-to method of choice. Theoretical projection based on the above single element apportioning of the multiplication step would be an ultimate 9 x9 x 4 =324x improvement in speed from the original script or the factor of 92 improvement over the reduced element multiplication in the current cython code, assuming GPU/CPU similarities in processing speed.   The remaining, strictly Python code is single threaded and would likely not be optimizable in the above fashion.

Overall Conclusions.   The Cython variant of the propagate script is an interesting, "intermediate" method for gaining speed in single threads without a computer/NVIDIA GPU combination capable of that version of processing.   A factor of nearly 1/10 reduction in processing time enables the Cython method to fall between the pyCUDA and standard Python simulation times.