WrightSim

Kyle Sunden

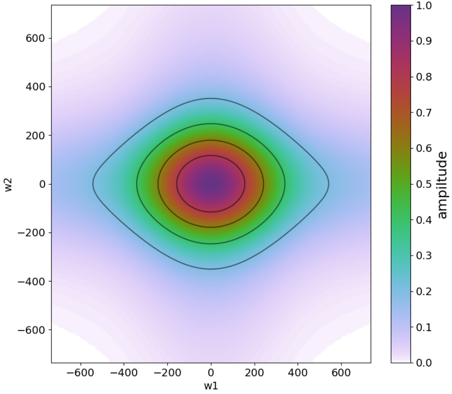
University of Wisconsin–Madison

January 3, 2018

Goals

- Reproduce experimental spectra *in silico*.
- Designed with experimentalists in mind.
- Uses numerical integration for flexibility, accuracy and interpretability.
- Focus on Frequency-domain spectroscopy, but techniques in principle extend to time-domain.
- Output retains frequency and phase information, can be combined with other simulations and measured similar to a monochromator.
- Selectivity in what portions of the overall signal are simulated, providing deeper understanding.

Sample Spectrum



Introduction

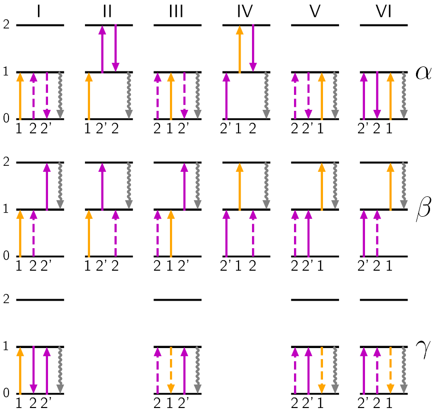
Presented here is a description of *what* is done to perform these simulations, to understand *why* it works, please refer to Kohler, Thompson, and Wright[1].

The simulation uses a set number of electric fields (3, in the case of the simulation presented here).

These electric fields interact in combinatorially large different fashions.

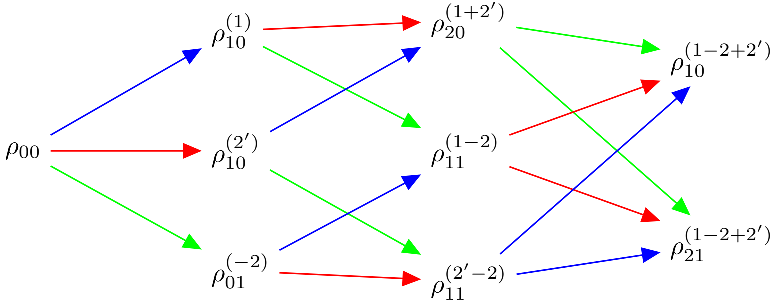
Interactions create superposition coherences and/or populations in material systems.

Pathways



There are six time orderings for interactions to occur (I-VI).
There are 16 independent pathways possible for two positive (solid up/dashed down) and one negative (dashed up/solid down) interactions.
Originally from [1].

Finite state automaton



$$\overline{\rho} \equiv \begin{bmatrix} \tilde{\rho}_{00} \\ \tilde{\rho}_{01}^{-2} \\ \tilde{\rho}_{10}^{2'} \\ \tilde{\rho}_{10}^{1} \\ \tilde{\rho}_{20}^{1+2'} \\ \tilde{\rho}_{11}^{1-2} \\ \tilde{\rho}_{11}^{2'-2} \\ \tilde{\rho}_{10}^{1-2+2'} \\ \tilde{\rho}_{21}^{1-2+2'} \end{bmatrix}$$

Density elements, encoded with quantum state (subscript) and the electric fields which have interacted (superscript). Colored arrows represent the different electric fields. All possible states which have the desired conditions for the process simulated are included. These form the state vector (right).[1]

Hamiltonian Matrix

$$\overline{\overline{Q}} \equiv \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -A_2 & -\Gamma_{10} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ A_{2'} & 0 & -\Gamma_{10} & 0 & 0 & 0 & 0 & 0 & 0 \\ A_1 & 0 & 0 & -\Gamma_{10} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & B_1 & B_{2'} & -\Gamma_{20} & 0 & 0 & 0 & 0 \\ 0 & A_1 & 0 & -A_2 & 0 & -\Gamma_{11} & 0 & 0 & 0 \\ 0 & A_{2'} & -A_2 & 0 & 0 & 0 & -\Gamma_{11} & 0 & 0 \\ 0 & 0 & 0 & 0 & B_2 & -2A_{2'} & -2A_1 & -\Gamma_{10} & 0 \\ 0 & 0 & 0 & 0 & -A_2 & B_{2'} & B_1 & 0 & -\Gamma_{21} \end{bmatrix}$$

Defines the transition between states, dependant on the electric field. Γ represents the dephasing/population decay. A and B variables incorporate the dipole moment and electric field terms.[1]

The dot product of this matrix and the density vector, $\overline{\rho}$, gives the change in in the density vector. This is repeated over many small time periods to achieve the recorded results.

NISE

NISE (Numerical Integration of the Schrödinger Equation) is an existing open-source implementation of the simulation for these kinds of spectra.[2]

It was written by Kohler and Thompson while preparing their manuscript.[1]

NISE uses a slight variation on the algorithm presented, which allows for a 7-element state vector, but requires two simulations.

The end result is the same.

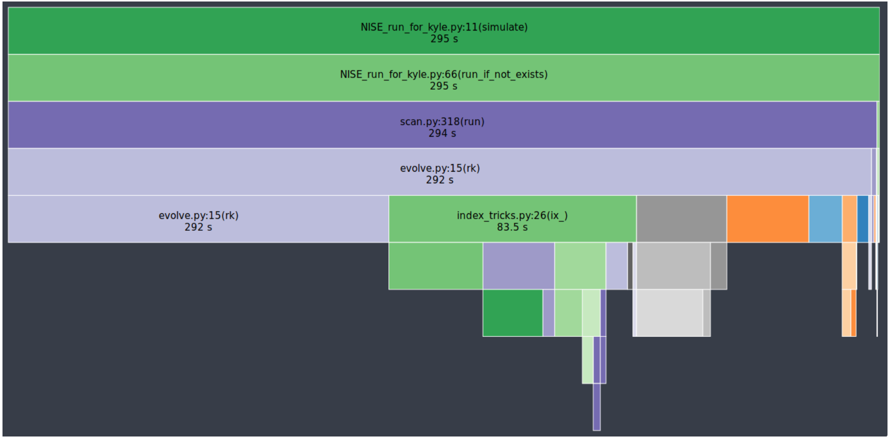
NISE is included as a reference for prior implementations.

Algorithmic Improvements

- Use single 9 x 9 matrix rather than two 7 x 7 matrices.
- 99.5% of time is spent in highly parallelizable loop.
- 1/3 of time is spent in a single function, `ix_`.
  - Removed entirely, in favor of a simpler communication of what to record.
- Significant time in `rotor` function which computes $cos(\theta) + i * sin(\theta)$.
  - Replaced with $exp(i * \theta)$, equivalent, more efficient, removed function call.
- Use variables to store and reuse redundant computations.

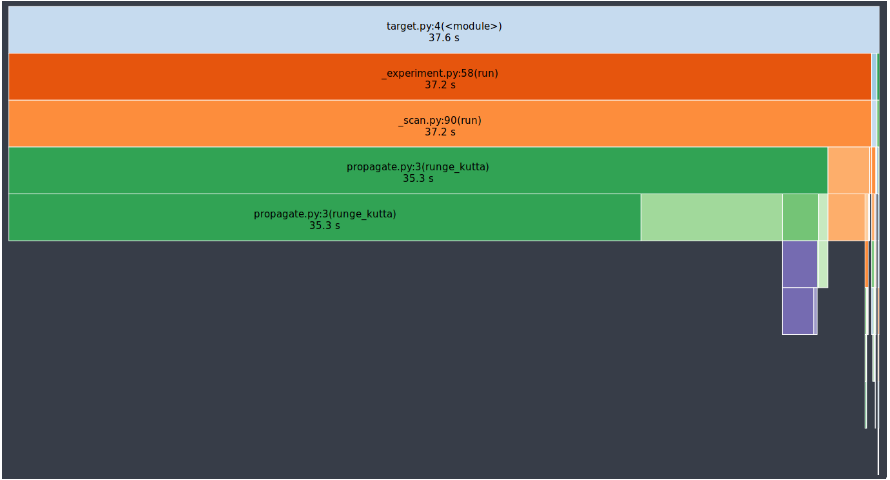
Resulted in almost an order of magnitude speed-up from algorithmic improvements alone. Remained highly parallelizable.

Profile trace of NISE



Python `cProfile` trace, Single Core implementation, visualized with SnakeViz.[3]

Profile trace of `WrightSim`

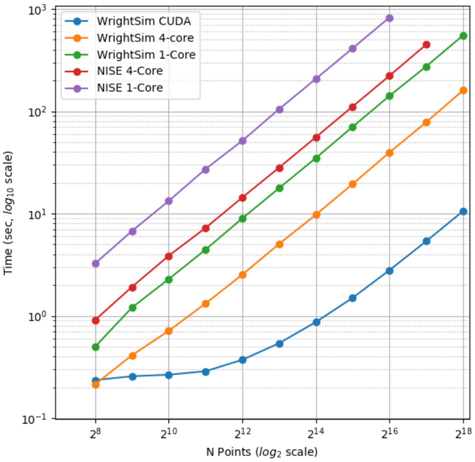


Python `cProfile` trace, Single Core implementation, visualized with SnakeViz.[3]

Parallel Implementations

- NISE already had CPU multiprocessed parallelism, using Python standard library interfaces.
  - `WrightSim` inherited this CPU parallel implementation
  - Results in a 4x speed-up on a 4-core machine, almost no reduction due to Amdahl's law.
- A new Nvidia CUDA [4] implementation.
  - Uses PyCUDA to call the kernel from within Python.
  - Just-in-time compiled (using nvcc) from C source code stored in Python strings.
  - Implementation differs slightly from pure Python implementation.
    - Only actively used Hamiltonians are held in memory, Python implementation computes all time steps ahead of time.
    - Similarly, only the actively used electric fields are held in memory.
    - Hand written dot product and vector addition, rather than the numpy implementations.

Scaling Analysis



Limitations

- For low number of points, the CUDA implementation is of limited use.
  - Around 200 ms required for compilation.
  - 4-Core multiprocessed becomes faster below approximately 256 points.
  - CUDA implementation currently uses a hard coded block size of 256.
  - Only multiples of 256 may be used at present to avoid illegal memory access.
- Independent CUDA simulations are memory limited.
  - only a certain amount of memory can be allocated for a single CUDA process.
  - Each point in the simulation requires 500 complex numbers (represented as doubles) to be allocated
  - Additional data is needed, but dominated by this array.
  - This array must be transferred back to the host.
  - The limit is between 2^{18} and 2^{19} points

Features

- Nonidealities like chirped pulses.
- Inhomogeneous samples
  - Results in broader peaks.
  - Can be modelled by translating a single computed response and adding.
- Measuring signal using Fourier transforms similar to how a monochromator selects a signal.
- Saving intermediate response values (Using HDF5 based file format)
- Saving compiled CUDA binary for more than one run at a time

Metaprogramming Features

Metaprogramming techniques are those which modify the code that is run as the program executes.
Just-in-time compilation enables many opportunities for metaprogramming.

- Using same-named functions to change the exact algorithms, while other calling functions do not need to be edited.
- Resolving shortcuts taken for statically allocated arrays.
- Potentially, entire C functions could be dynamically generated by inspecting the Python code, resulting in new Hamiltonians only being required to be implemented once, in Python.

Usability

One important thing to do is to take a step back and think about how users interact with the program.

- Ensure the API is sensible, easy to follow, and well documented.
- Provide ways of configuring via configuration files instead of code.
- Think about implementing a GUI interface, targeted to experimentalists.
- Implement new Hamiltonians.
- Ensure code is robust, with proper values being transferred to and from the CUDA Device with different Hamiltonian instances.
- Resolve hard-coded initial values.

Algorithmic Improvements

- TODO

Conclusions

- TODO

Daniel D. Kohler, Blaise J. Thompson, and John C. Wright. "Frequency-domain coherent multidimensional spectroscopy when dephasing rivals pulsewidth". In: *The Journal of Chemical Physics* 147.8 (2017), p. 084202. DOI: 10.1063/1.4986069. URL: https://doi.org/10.1063%2F1.4986069.

Wright Group. *NISE: Numerical Integration of the Shrödinger Equation*. 2016. URL: http://github.com/wright-group/NISE.

jiffyclub. *SnakeViz*. 2017. URL: http://jiffyclub.github.io/snakeviz/.

John Nickolls et al. "Scalable parallel programming with CUDA". In: *Queue* 6.2 (2008), p. 40. DOI: 10.1145/1365490.1365500. URL: https://doi.org/10.1145%2F1365490.1365500.