WrightSim Documentation

Release 0.1.0

WrightSim Developers

Sep 19, 2023

CONTENTS:

CHAPTER

ONE

MULTIPROCESSING IN WRIGHTSIM

WrightSim provides three different levels of computation:

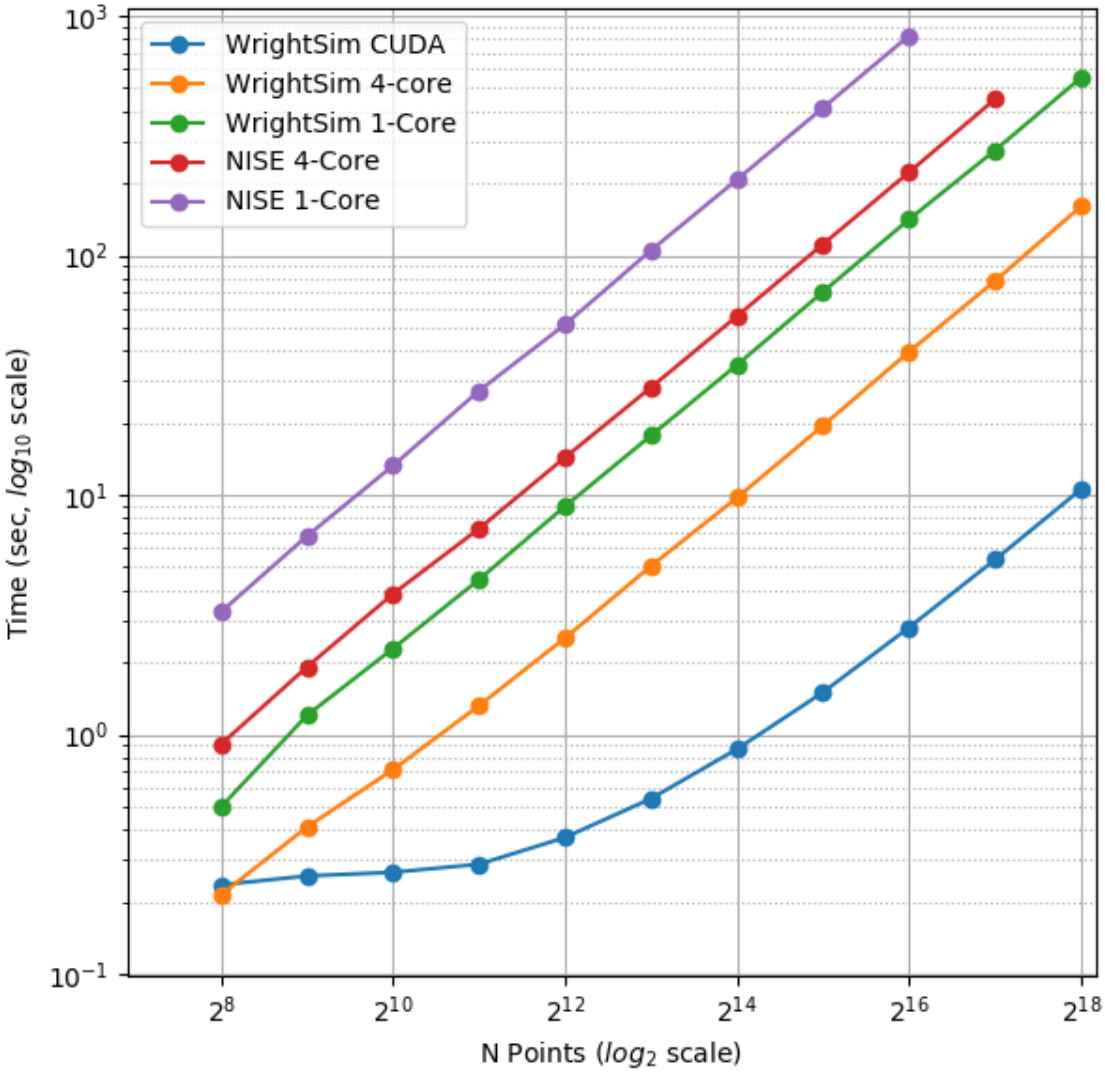
- Single Core CPU: Run single threaded
- Multi Core CPU: Use multiprocessing to run on as many cores as the CPU has available.
- CUDA GPU: Use PyCUDA to run a highly parallel implementation.

Caution: Windows Users

Using multiprocessing on Windows requires using `if __name__ == "__main__":` to prevent infinite processes from being spawned. This does not apply to unix-styled systems.

Additionally, the CUDA multiprocessing has not been tested under Windows.

1.1 Scaling Analysis



Both NISE and WrightSim scale linearly with respect to the number of points. There is a 4x speed up when running on 4 CPU cores. The CUDA implementation is 10x faster than 4 CPU cores, though it incurs a constant 200 millisecond offset for compilation.

CHAPTER

TWO

PROPAGATION METHODS

Propagation methods are the heart of the computation which do the numerical integration of the differential equations.

Python propagators satisfy the signature:

```
def propagator_name(t, efields, n_recorded, hamiltonian)
   pass
```

In CUDA C, the propegators have a different signature, due to limited memory constraints:

```
__device__
pycuda::complex<double>* propagator_name(
                                        const double time_start,
                                        const double time_end,
                                        const double dt,
                                        const int nEFields,
                                        double* efparams,
                                        int* phase_matching,
                                        const int n_recorded,
                                        Hamiltonian ham,
                                        pycuda::complex<double> *out
                                       )
```

Note that due to the memory constrained nature of the CUDA device code, time and efields are passed as parameters rather than the arrays themselves. The Hamiltonian is a struct, defined by each Hamiltonian object that supports CUDA. It must include fields for `int nStates`, `int nRecorded`, and `int* recorded_indices`, as well as any fields used in the Hamiltonian-defined `Hamiltonian_matrix` method. Also note that the output is pre-allocated and passed in by reference.

2.1 Runge-Kutta

Currently the only propagation method implemented in `WrightSim`. The version implemented here is a second order Runge-Kutta technique. It is available for both the Python and CUDA implementations.

CHAPTER

THREE

WRIGHTSIM

3.1 WrightSim package

3.1.1 Subpackages

WrightSim.driven package

Module contents

WrightSim.experiment package

Module contents

WrightSim.experiment.**builtin**(*name*)

WrightSim.experiment.**from_ini**(*p*)

WrightSim.hamiltonian package

Submodules

WrightSim.hamiltonian.TRSF_default module

class WrightSim.hamiltonian.TRSF_default.**Hamiltonian**(*rho=None*, *tau=None*, *mu=None*, *omega=None*, *labels=['gg', 'Ig_1', 'Ig_2', 'ig_1', 'ig_22I,g', '2i,g', 'c,g', 'ag', 'bg']*, *time_orderings=[1, 2, 3]*, *phase_cycle=False*, *propagator=None*)

Bases: object

matrix(*efields*, *time*)

WrightSim.hamiltonian.default module

class WrightSim.hamiltonian.default.**Hamiltonian**(*rho=None*, *tau=None*, *mu=None*, *omega=None*, *w_central=7000.0*, *coupling=0*, *propagator=None*, *phase_cycle=False*, *labels=['00', '01 -2', "10 2'", '10 1', "20 1+2'", '11 1-2', "11 2'-2", "10 1-2+2'", "21 1-2+2'"]*, *time_orderings=[1, 2, 3, 4, 5, 6]*, *recorded_indices=[7, 8]*)

Bases: object

cuda_matrix_source = "\n /**\n * Hamiltonian_matrix: Computes the Hamiltonian matrix for an indevidual time step.\n * NOTE: This differs from the Python implementation, which computes the full time \n * dependant hamiltonian, this only computes for a single time step\n * (to conserve memory).\n * \n * Parameters\n * ----------\n * Hamiltonian ham: A struct which represents a hamiltonian,\n * containing orrays omega, mu, and Gamma\n * cmplx* efields: A pointer to an array containg the complex valued\n * electric fields to use for evaluation\n * double time: the current time step counter\n *\n * Output\n * -------\n * cmplx* out: an N x N matrix containing the transition probabilities\n *\n */\n __device__ void Hamiltonian_matrix(Hamiltonian ham, pycuda::complex<double>* efields,\n double time, pycuda::complex<double>* out)\n {\n // Define state energies\n double wag = ham.omega[1];\n double w2aa = ham.omega[8];\n\n // Define dipoles\n //TODO: don't assume one, generalize\n pycuda::complex<double> mu_ag = 1.;//ham.mu[0];\n pycuda::complex<double> mu_2aa = 1.;//ham.mu[1];\n\n // Define the electric field values\n pycuda::complex<double> E1 = efields[0];\n pycuda::complex<double> E2 = efields[1];\n pycuda::complex<double> E3 = efields[2];\n\n // Define helpful variables\n pycuda::complex<double> A_1 = 0.5 * I * mu_ag * E1 * pycuda::exp(-1. * I * wag * time);\n pycuda::complex<double> A_2 = 0.5 * I * mu_ag * E2 * pycuda::exp(I * wag * time);\n pycuda::complex<double> A_2prime = 0.5 * I * mu_ag * E3 * pycuda::exp(-1. * I * wag * time);\n pycuda::complex<double> B_1 = 0.5 * I * mu_2aa * E1 * pycuda::exp(-1. * I * w2aa * time);\n pycuda::complex<double> B_2 = 0.5 * I * mu_2aa * E2 * pycuda::exp(I * w2aa * time);\n pycuda::complex<double> B_2prime = 0.5 * I * mu_2aa * E3 * pycuda::exp(-1. * I * w2aa * time);\n\n //TODO: zero once, take this loop out of the inner most loop\n for (int i=0; i<ham.nStates * ham.nStates; i++) out[i] = pycuda::complex<double>();\n\n // Fill in appropriate matrix elements\n if(ham.time_orderings[2] || ham.time_orderings[4])\n out[1*ham.nStates + 0] = -1. * A_2;\n if(ham.time_orderings[3] || ham.time_orderings[5])\n out[2*ham.nStates + 0] = A_2prime;\n if(ham.time_orderings[0] || ham.time_orderings[1])\n out[3*ham.nStates + 0] = A_1;\n if(ham.time_orderings[2])\n out[5*ham.nStates + 1] = A_1;\n if(ham.time_orderings[4])\n out[6*ham.nStates + 1] = A_2prime;\n if(ham.time_orderings[3])\n out[4*ham.nStates + 2] = B_1;\n if(ham.time_orderings[5])\n out[6*ham.nStates + 2] = -1. * A_2;\n if(ham.time_orderings[0])\n out[4*ham.nStates + 3] = B_2prime;\n if(ham.time_orderings[1])\n out[5*ham.nStates + 3] = -1. * A_2;\n if(ham.time_orderings[1] || ham.time_orderings[3])\n {\n out[7*ham.nStates + 4] = B_2;\n out[8*ham.nStates + 4] = -1. * A_2;\n }\n if(ham.time_orderings[0] || ham.time_orderings[2])\n {\n out[7*ham.nStates + 5] = -2. * A_2prime;\n out[8*ham.nStates + 5] = B_2prime;\n }\n if(ham.time_orderings[4] || ham.time_orderings[5])\n {\n out[7*ham.nStates + 6] = -2. * A_1;\n out[8*ham.nStates + 6] = B_1;\n }\n\n // Put Gamma along the diagonal\n for(int i=0; i<ham.nStates; i++) out[i*ham.nStates + i] = -1. * ham.Gamma[i];\n }\n"

cuda_mem_size = 64

```
cuda_struct = '\n #include <pycuda-complex.hpp>\n #define I
pycuda::complex<double>(0,1)\n\n struct Hamiltonian {\n int nStates;\n int nMu;\n
int nTimeOrderings;\n int nRecorded;\n pycuda::complex<double>* rho;\n
pycuda::complex<double>* mu;\n double* omega;\n double* Gamma;\n\n char*
time_orderings;\n int* recorded_indices;\n };\n '
```

matrix(*efields*, *time*)

Generate the time dependant Hamiltonian Coupling Matrix.

Parameters

- **efields** (*ndarray<Complex>*) – Contains the time dependent electric fields. Shape (M x T) where M is number of electric fields, and T is number of timesteps.
- **time** (*1-D array <float64>*) – The time step values

Returns
Shape T x N x N array with the full Hamiltonian at each time step. N is the number of states in the Density vector.

Return type
ndarray <Complex>

to_device(*pointer*)

Transfer the Hamiltonian to a C struct in CUDA device memory.

Currently expects a pointer to an already allocated chunk of memory.

Module contents

WrightSim.mixed package

Submodules

WrightSim.mixed.propagate module

WrightSim.mixed.propagate.**runge_kutta**(*t*, *efields*, *n_recorded*, *hamiltonian*)

Evolves the hamiltonian in time using the runge_kutta method.

Parameters

- **t** (*1-D array of float*) – Time points, equally spaced array. Shape T, number of time-points
- **efields** (*ndarray <Complex>*) – Time dependant electric fields for all pulses. SHape M x T where M is number of electric fields, T is number of time points.
- **n_recorded** (*int*) – Number of timesteps to record at the end of the simulation.
- **hamiltonian** (Hamiltonian) –

  The hamiltonian object which contains the inital conditions and the
  function to use to obtain the matrices.

Returns
ndarray – 2-D array of recorded density vector elements for each time step in n_recorded.

Return type
<Complex>

WrightSim.mixed.system module

Module contents

3.1.2 Submodules

3.1.3 WrightSim.integration module

class WrightSim.integration.**Response**(**args*, ***kwargs*)

Bases: Data

3.1.4 WrightSim.measure module

class WrightSim.measure.**Spectrum**(*axes*, *constants=[]*)

Bases: object

measure(**args*)

save()

3.1.5 WrightSim.response module

class WrightSim.response.**Response**(**args*, ***kwargs*)

Bases: Data

save(*p=None*)

Save as root of a new file.

Parameters

- **filepath** (*Path-like object (optional)*) – Filepath to write. If None, file is created using natural_name.
- **overwrite** (*boolean (optional)*) – Toggle overwrite behavior. Default is False.
- **verbose** (*boolean (optional)*) – Toggle talkback. Default is True

Returns

Written filepath.

Return type

str

WrightSim.response.**load**(*p*)

3.1.6 Module contents

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

W

INDEX