TDSE Code - User Guide, Best Practices, and Developers Notes

J. Venzke

October 30, 2017

1 Introduction

The TDSE code discussed here was mainly developed by Joel Venzke while working on his Ph.D. in the Ultrafast Theory Group in JILA. Cory Goldsmith and other group members have also contributed to this code in various forms. It solves the Time Dependent Schödinger Equation (TDSE) for various targets in a laser field with for single active electrons. It has support for two or more active electrons, but the computational power of modern super computers limits the calculation that are possible to 2 electrons in 2 spacial dimensions. For single active electrons, I have developed a cylindrical 2D code for linear polarized light. All other calculations requirer use of the Cartesian grid code. Future work will look to push towards a 2 electron full 6D code, and extending the code to different coordinate systems/basis methods.

1.1 Code Overview

At the attosecond ($10^{-18} seconds$) time scale, molecular motion is frozen, and electron dynamics dominate all processes. In order to study how electrons interact with attosecond laser pulses, we solve the Time-Dependent Schödinger Equation (TDSE). This can be done relatively easily for 1 electron with linear polarized light, however, circular polarization and multi-electron effects make the calculations more computationally interesting.

This code has support for real space propagation of the TDSE using finite differences for spacial derivatives and Crank-Nicolson for time propagation. The codes has been shown to scale out to 6,000+ processors (we ran out of computer). It can also simulate interesting physics on a laptop if you have a few days of free time. The main drive for developing this code is to push simulations passed the current limit.

1.2 Development Goals

This code is being designed with large scale numeric simulation in mind. Having a code that is capable of running simulations out of the reach of many groups allows a user to push the edge of our scientific knowledge. That being said, for small scale simulation, there are better more efficient codes. However, those codes will reach limitations due to hardware which is where a large scale code like this starts to shine. For that reason, MPI is used for parallelization so that we can utilize HPC systems.

Since this code is for physics, we want the users to be able to focus on physics rather than the computer science back end. As a result, all simulations only require an input file with no need to write code. It maybe necessary to learn python for custom data visualizations, but many of the visualizations needed to analysis the results are included with the repo.

It would also be nice that when the simulation is complete, the code produces enough plots to make sure that the simulation worked and a basic overview of the interesting physics the simulation contains. For this reason, all visualization is done in a batch mode and images are saved in a “figs” directory.

Finally, we want this code be useful in future iterations. Therefor we used GitHub for version control. We are also working on in source documentation to ease development efforts.

2 Theory

The TDSE can be written simply as

$$i\frac{\partial}{\partial t}\psi(x,t) = \hat{H}\psi(x,t) \tag{1}$$

In the velocity gauge the Hamiltonian becomes

$$\hat{H} = \sum_e \left(\frac{\hat{\mathbf{p}}_e^2}{2} - \frac{\mathbf{A}(t)\cdot\hat{\mathbf{p}}_e}{c} - \sum_n \frac{Z_n}{r_{e,n}} \right) + \sum_{e_1<e_2} \frac{1}{r_{e_1,e_2}} \tag{2}$$

- $\hbar = e = m = 1$ (atomic units)
- $\hat{H}$ is the Hamiltonian
- $\hat{\mathbf{p}}$ is the momentum operator $(-i\nabla)$
- $\mathbf{A}(t)$ is the vector potential that describes the laser
- c is the speed of light (137ish in a.u.)
- Z_n is the nuclear charge
- $r_{i,j}$ is the Euclidean distance between i and j

This Hamiltonian assumes that the wavelength is much larger than the radius of the atom (dipole approximation), the field has a large number of photons (it treats fields classically), and the nuclei are fixed in space during the simulation (molecular motion is much slower than electron motion). Moving (classically and quantum mechanically) nuclei are on the wish list for this code.

The first term in Equation 2 is the kinetic energy of the electron. In atomic units, this can be written such that

$$\frac{\hat{\mathbf{p}}_e^2}{2} = \frac{\nabla_e^2}{2} \tag{3}$$

Note that the subscript ∇_e^2 means the derivatives only act on the eth electron and acts like the identity operator on all other electrons.

The second term is the laser electron interaction.

$$-\frac{\mathbf{A}(t)\cdot\hat{\mathbf{p}}_e}{c} = \frac{\mathbf{A}(t)\cdot i\nabla_e}{c} \tag{4}$$

The laser is given as a vector potential. A short discussion on that is provided in Section 2.4.1.

The third term is the Coulomb potential for each nuclei.

$$-\sum_n \frac{Z_n}{r_{e,n}} \tag{5}$$

This is often implemented with a soft core (Section 2.3.1) to avoid the singularity at $r_{n,e} = 0$. The code allows for any locations for atomic targets. This allows any molecule to be implemented in the code without the need to recompile. You can also add frozen electrons to a nuclei using single active electron potentials found in Section 2.3.3. This is one of term that makes this equation hard hard to solve.

The last term is the electron electron coloration term.

$$\sum_{e_1<e_2} \frac{1}{r_{e_1,e_2}} \tag{6}$$

It gives the repulsion of the electron with all other electrons. This is the term that couples every electron in the system and leads to an N electron calculation becoming a 3N dimensional Hilbert space leading to 2 electron simulations being the biggest we can hope to simulate with current computing technologies. If you neglect this term and put it into some effective potential, you can get either a TDDFT type calculation or a single active electron potential depending on how you do it.

2.1 Spacial Derivatives

For spacial derivatives, this code utilized finite differences. When using finite differences, derivatives can represented by a set of coefficients called a stencil. The stencil provides the non-zero c_i weights for various sample points of the function. You can then write the second derivative of ψ known at various evenly spaced grid points labeled by n as

$$\frac{d^2}{dx^2}\psi_n = \frac{1}{\Delta x^2}\sum_i c_i \psi_i \tag{7}$$

The non zero c_i coefficients are given in the table below for various orders of accuracy.

Order	c_{n-3}	c_{n-2}	c_{n-1}	c_n	c_{n+1}	c_{n+2}	c_{n+3}
2nd			-1	2	-1		
4th		$-\frac{1}{12}$	$\frac{4}{3}$	$-\frac{5}{2}$	$\frac{4}{3}$	$-\frac{1}{12}$	
6th	$\frac{1}{90}$	$-\frac{3}{20}$	$\frac{3}{2}$	$-\frac{49}{18}$	$\frac{3}{2}$	$-\frac{3}{20}$	$\frac{1}{90}$

For the remainder of this discussion, we will assume we are using second order derivatives. However, this will be expendable to higher order derivatives by adding more off diagonal elements in matrices we will discuss in this section.

For the start of this discussion, we will consider a 1D wavefunction. We will later extend this to ND wavefunctions. We start with $\psi(x)$ known at various points along the x axis spaced by a grid step of dx. We will label the points by n such that $\psi_n = \psi(x_0 + dx * n)$ with x_0 being the lowest x value. Plugging this into Equation 7 we get

$$\frac{d^2}{dx^2}\psi_n = \frac{1}{\Delta x^2}\left(-\psi_{n-1} + 2\psi_n - \psi_{n+1}\right) \tag{8}$$

Now we can write this for the point ψ_{n+1} giving us

$$\frac{d^2}{dx^2}\psi_{n+1} = \frac{1}{\Delta x^2}\left(-\psi_n + 2\psi_{n+1} - \psi_{n+2}\right) \tag{9}$$

This can be done until we hit the other end of the grid. If we take the boundary condition that $\psi(x_0 - dx) = 0$ and likewise on the other end, we can write our operator as a system of linear equations in the form of a matrix. Our matrix becomes

$$\frac{d^2\psi}{dx^2} = \frac{1}{\Delta x^2}\begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix}\begin{bmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{N-1} \\ \psi_N \end{bmatrix} \tag{10}$$

2.2 Time Propagation

2.2.1 Exterior Complex Scaling (ECS)

2.3 Atoms and Molecules

These are the various targets that this code supports. Each one will only be supported for certain solvers, so please double check which ones are implemented. Validation will eventually be added to the Parameters class

A two active electron target of Helium. Currently being developed using finite differences.

2.3.1 Soft Cores

2.3.2 Hydrogen Like

2.3.3 Single Active Electron

2.4 Lasers

2.4.1 Electric field vs Vector potential

3 Build System

The build system changes from time to time. For the most up to date version, see the README in the root directory of this repository.

4 Input

All input values are in atomic units. The input file name should be "input.json"

4.1 Parameters

The following is the base set of parameters. Those with sub parameters will have additional sub sections. An example input file can be found in Section 4.4.

- delta_t
  - The size of the time step in atomic units.
- dimensions
  - A list containing information about each dimension see Section 4.2 for more details.
- restart
  - When set to 1, it will restart the simulation from the last checkpoint in the hdf5 file. It also checks to see if the input files match the original one. Set to 0 if you want a new simulation.
- target
  - Tells the code what target the laser will be incident on. See Sections 2.3 for added information and the list of supported targets.
- gobbler
  - The percent of the grid that is "gobbled" by the absorbing potential so the wavefunction does not reflect off the edge of the box.
- pulses
  - A list of pulses used as a super position to create the laser field. See Section 4.3 for more information.

4.2 Dimensions

Each dimension requires the following two input parameters.

- dim_size
  - The length of that dimension in atomic units.
- delta_x
  - The step sizes in that dimension in atomic units.

4.3 Pulses

Each pulse requires the following input parameters.

- pulse_shape
  - The shape of the pulse envelope. Currently supports sin^2 envelopes.
- cycles_on
  - The number of cycles the pulse uses to turn on.
- cycles_plateau
  - The number of cycles the pulse is at max amplitude. Usually 0.0
- cycles_off
  - The number of cycles the pulse uses to turn off.
- cycles_delay
  - The number of cycles before the pulse starts to turn on.
- cep
  - The carrying phase envelope of the pulse. It is defined at the time the pulse starts to turn on.
- energy
  - The fundamental angular frequency of the pulse. Corresponds to the energy of the photons in atomic units.
- e_max
  - The maximum amplitude of the pulse in atomic units.

4.4 Example File

Input files are in json format. Here is an example input file:

```
{
  "delta_t": 0.2,
  "dimensions": [
    {"dim_size": 1.0,
    "delta_x":   0.01},
    {"dim_size": 2.05,
    "delta_x":   0.5}],
  "restart": 0,
  "target":  "He",
  "gobbler": 0.9,
  "pulses": [
    {"pulse_shape":   "sin2",
      "cycles_on":      3.0,
      "cycles_plateau": 1.0,
      "cycles_off":     3.0,
      "cycles_delay":   0.0,
      "cep":            0.0,
      "energy":         5.338e-3,
      "e_max":          1.0},
    {"pulse_shape":   "sin2",
```

```
      "cycles_on":      3.0,
      "cycles_plateau": 0.0,
      "cycles_off":     3.0,
      "cycles_delay":   1.0,
      "cep":            0.0,
      "energy":         5.338e-3,
      "e_max":          1.0}]
}
```

5 Classes

The classes used in this code are written so they can be tested independently. Therefore, all classes that depend on other classes require those classes to be passed into the constructor. The developer is responsible to not delete these classes until other dependent classes are deleted.

For the most part, the classes do what you would expect. However, there are small deviations like the HDF5Wrappers writing the Parameters to the file rather than the other way around. This is done to enable restarts to not delete old datasets.

5.1 PETSCWrapper

This class takes care of setting up SLEPC, PETSC, and MPI. It comes first so that the finalize calls are made once every other class has cleaned up after its self.

5.2 ViewWrapper

This class wraps the HDF5 interface that PETSC supplies. It is used for dumping the wavefunctions to disk.

5.3 Parameters

This class holds the input file information and will be used by all other classes. It depends on a json parser (https://github.com/nlohmann/json) and reads inputs in json format. For more information about input parameter definitions and how they are used, see Section 4. This class is mainly used as a data structure. That being said, it does provided tools to validate various input parameters to limit user error. It is also used to convert the various laser input styles to the "default" style that the Pulse class needs to create the laser fields.

The code currently reads the input files from every processor in use. This hasn't been an issue running on upwards of 6,000 processors, but it should be changed at some point.

5.4 HDF5Wappers

IO for this code is done using HDF5 files. HDF5 files provide the ability to utilize parallel file systems, generate self describing data files, and write compressed data. With these advantages, it becomes slightly more difficult to write data. To alleviate this issues, this class provides tools for writing to HDF5 files in a straight forward way.

Writing multidimensional arrays is difficult in HDF5. To avoid the added headache, one denominational arrays with the access pattern of

$$array[x + y * nx] = array[x][y] \tag{11}$$

and

$$array[x + y * nx + z * nx * ny] = array[x][y][z] \tag{12}$$

and so on. Note that the ordering flips when using python to visualize the data due to the c vs fortran ordering of arrays. Be sure to pay close attention to axis labels to make sure the data is orientated correctly.

5.5 Pulse

Pulse handles creating and storing various parts of the pulse. Much of the functionality is private rather than public to prevent accessing data that is not allocated. Since storing each individual pulse is only useful until it is printed to a file, it is deallocated inside the constructor. The result is a much more efficient use of memory, and protection from accessing garbage arrays. See Section 4 on how to define pulses.

5.6 Hamiltonian

We are interested in solving the Solving the Time-Dependent Schödinger Equation for atoms and molecules.

$$i\frac{\partial}{\partial t}\psi(x,t) = \hat{H}\psi(x,t) \tag{13}$$

Which leads to a Hamiltonian of

$$\hat{H} = \sum_e \left(\frac{\hat{\mathbf{p}}_e^2}{2} - \frac{\mathbf{A}(t)\cdot\hat{\mathbf{p}}_e}{c} - \sum_n \frac{Z_n}{r_{e,n}} \right) + \sum_{e_1<e_2} \frac{1}{r_{e_1,e_2}} \tag{14}$$

The full problem is pretty much impossible to solve for 3 or more electrons. For a two electron atom (He like), we have the following Hamiltonian.

$$\hat{H} = \frac{\hat{p_1}^2}{2} + \frac{\hat{p_2}^2}{2} - \frac{\vec{A}\cdot(\hat{p_1}+\hat{p_2})}{c} - \frac{Z}{x_1} - \frac{Z}{x_2} + \frac{1}{r_{12}} \tag{15}$$

The Hamiltonian class takes the input file and defines the appropriate matrix to operate on our wave function. There is tons of index gymnastics going on in this class. Creation of the Hamiltonian is currently around 30% of the total run time of a simulation. This code has some low hanging optimizations like storing the time independent Hamiltonian. This should be the next priority in optimizations. For a full explanation the numeric methods we use and justification for various tools, see Section 2.

5.7 Simulation

The simulation class implements various propagation and eigen state calculations. It then utilized every other class to provide the appropriate IO and behavior at various points in the simulation.

6 Testing

There is very little testing used in this code. If you would like to work on this or notices something is wrong. Please let me know.

7 Wish list

- Faster algorithm (RBF, radial code, non uniform grid)
- Free propagation support (Density Matrix)
- New features (fast two electron code)
- Moving nuclei (classical and quantum mechanical)

8 Supporting works

8.1 Posters

8.2 Talks

8.3 Papers