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

J. Venzke

July 23, 2019

## Contents

# 1 Introduction

## 1.1 Code Overview

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.

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 hours of free time. The main drive for developing this code is to push simulations passed the current limit.

## 1.2 Problems of interest

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. Gaining a deeper understanding the behavior of electrons induced by intense ultrashort laser pulses is the reason this code was developed.

## 1.3 Development Goals

This code is being designed with large scale numeric simulation in mind. We want to be limited by the largest computer we can get our hands on, not the fastest desktop on the market. 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, the pipe dream is to have 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. You may also need to implement small c++ changes if you wish to simulate targets that do not fit the current layout or various other unsupported tasks, though I hope to make this code relatively general.

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.

## 1.4 Key Features

TODO

## 1.5 Recommended Programing Background

It is recommended you are familiar with python including the numpy, matplotlib, and h5py modules. This will allow you to make changes to the analysis software and analyze the results of the code beyond what already exists. It is also recommended you are familiar with the command prompt (terminal). Useful knowledge includes general bash commands such as cd, mkdir, sed, grep, chmod, and how to execute a

binary file. There are other commands such as mpiexec, h5ls, and h5dump that are useful to know, but will be talked about when needed in this documentation. You should also become familiar with any schedulers you may need to use on the various clusters and super computers you plan to use.

If you need to update the main code, you will need knowledge of C++. Most updates at this point are standard C++ code, however, the PETSc and SLEPc libraries are used for any matrix vector operations, and the IO is preformed through the HDF5 library.

# 2 Installation

Compiling the TDSE library itself is easy. However, some of the dependencies are tricky to get installed. Here is a short overview with some tips and tricks to help you out. Please read the If you run into issues, shoot me an email.

The steps required to install this code are as follows. More detailed directions are in the following subsections.

1. Choose the compiler you plan on using
2. Check module system/package manager for dependencies
3. Install HDF5 (special version required, C++ and MPI)
4. Install PETSC (link HDF5)
5. Install SLEPC
6. Install TDSE library
7. Update your bashrc file

## 2.1 Compilers

This code has been installed with gcc and intel compilers with c++11 or better support. Portland group compilers have not been tested, but should work. If you are using a Mac, install gcc with the homebrew package manager (https://brew.sh) instead of using clang as you will need a fortran compiler for PETSC.

## 2.2 Dependencies

Before beginning the install process, make sure the following software is installed on your computer. Make sure they are compiled with the compiler you intend on using to compile the TDSE library. You should find them on the module system at most HPC centers and many local clusters. Here is the list of packages to look for:

- boost
- boost-mpi
- cmake
- gsl
- open-mpi (or equivalent)
- zlib (can be installed by HDF5)

If one or more of these packages are not installed, you will need to do so manually. For Mac users, I recommend using homebrew (https://brew.sh) as your package manager and start by installing gcc followed by the rest of the packages. I have not had to install these packages on any other system, so if you need to, talk to a system admin or try your hand at installing them off their website.

## 2.3 HDF5

1. Download and untar HDF5 source (https://www.hdfgroup.org)
2. Make the directory you wish to install HDF5 in
3. cd to the directory you just make
4. load any modules for the dependencies (if needed)
5. Run the script at end of this section after these changes
   - Update ${PATH_TO_HDF5_SOURCE} to where the hdf5 source you downloaded is located
   - Update ${PATH_TO_HDF5_INSTALL} to where you want your compiled version of HDF5 to live
6. If the test pass, HDF5 is good to go

———- BEGIN FILE ———

```
#!/bin/bash

# run any module loads you need (from dependencies)

# configure HDF5
${PATH_TO_HDF5_SOURCE}/configure \
  CC=mpicc \
  FC=mpif90 \
  CXX=mpic++ \
  --prefix=${PATH_TO_HDF5_INSTALL} \
  --enable-build-mode=production \
  --disable-dependency-tracking \
  --enable-static=yes \
  --enable-shared=yes \
  --enable-unsupported \
  --enable-cxx \
  --enable-fortran \
  --enable-parallel

# run the following make commands
make
make check
make install
```

———- End FILE ———

## 2.4 PETSC

1. Download and untar PETSC source (https://www.mcs.anl.gov/petsc/)
2. cd to the directory with the petsc source
3. load any modules for the dependencies (if needed)
4. Run the script at end of this section after these changes
   - update the script to load any modules (if needed)
   - remove anaconda from your path
   - Update ${PATH_TO_PETSC_INSTALL} to where you want your compiled version of PETSC to live

- Update ${PATH_TO_HDF5_INSTALL} to where you want your compiled version of HDF5 to live

5. Run the make commands that are produced by the petsc build script output.
6. If the test pass, PETSC is good to go

———- BEGIN FILE ———

```
#!/bin/bash
export PETSC_DIR=‘pwd‘

# run any module loads you need (from dependencies)

# remove anaconda from your path if installed (it can cause issues with the PETSC build)
# example code to remove anaconda3 from path
# export PATH=$(echo "$PATH" | sed -e ’s/:\/Users\/username\/anaconda3\/bin//’)
# export PATH=$(echo "$PATH" | sed -e ’s/\/Users\/username\/anaconda3\/bin://’)

# configure PETSC
./configure \
  CXXOPTFLAGS="-O2" \
  COPTFLAGS="-O2" \
  FOPTFLAGS="-O2" \
  --prefix=${PATH_TO_PETSC_INSTALL} \
  --with-shared-libraries=1 \
  --with-pthread=0 \
  --with-openmp=0 \
  --with-debugging=0 \
  --with-ssl=0 \
  --with-x=0 \
  --with-valgrind=1 \
  --with-fortran-kernels=1 \
  --with-cxx-dialect=c++11 \
  --with-scalar-type=complex \
  --with-hdf5-dir=${PATH_TO_HDF5_INSTALL} \
  --download-fftw=yes \
  --download-superlu_dist=yes \
  --download-superlu=yes \
  --download-suitesparse=yes \
  --download-metis=yes \
  --download-parmetis=yes \
  --download-scalapack=yes \
  --download-mumps=yes

# run the make commands that print at the end of the configure script if successful
# dont forget to load modules
```

———- End FILE ———

## 2.5 SLEPC

1. Download and untar SLEPC source (http://slepc.upv.es)
2. cd to the directory with the slepc source
3. ensure ${PETSC_DIR} is set to ${PATH_TO_PETSC_INSTALL} by running
   “export PETSC_DIR=${PATH_TO_PETSC_INSTALL}”
   (without the quotes and replacing ${PATH_TO_PETSC_INSTALL} as done before)

4. load any modules for the dependencies (if needed)
5. Run the script at end of this section after these changes
    - update the script to load any modules (if needed)
    - Update ${PATH_TO_SLEPC_INSTALL} to where you want your compiled version of SLEPC to live
6. Run the make commands that are produced by the petsc build script output.
7. If the test pass, SLEPC is good to go

———- BEGIN FILE ———

```
#!/bin/bash
export SLEPC_DIR=‘pwd‘

# run any module loads you need (from dependencies)

# configure SLEPC
./configure \
  --prefix=${PATH_TO_SLEPC_INSTALL}

# run the make commands that print at the end of the configure script if successful
```

———- End FILE ———

## 2.6 TDSE

1. Clone the TDSE directory (https://github.com/Joel-Venzke/TDSE)
2. Make a “bin” directory inside the TDSE repo
3. Make a “build” file inside the TDSE repo (see file near end of subsection as example)
4. Create a “Makefile” in the “TDSE/src” directory (see example near end of subsection)
    - You may need to update the
5. Ensure ${PETSC_DIR} is set to ${PATH_TO_PETSC_INSTALL} by running
   “export PETSC_DIR=${PATH_TO_PETSC_INSTALL}”
   (without the quotes and replacing ${PATH_TO_PETSC_INSTALL} as done before)
6. Ensure ${SLEPC_DIR} is set to ${PATH_TO_SLEPC_INSTALL} by running
   “export SLEPC_DIR=${PATH_TO_SLEPC_INSTALL}”
   (without the quotes and replacing ${PATH_TO_SLEPC_INSTALL} as done before)
7. cd to the TDSE repo’s root directory
8. run ./build
9. If the code compiles correctly, the binary should appear as “TDSE/bin/TDSE” and you are done

———- BEGIN build ———

```
#!/bin/bash

# run any module loads you need (from dependencies)

# remove anaconda from your path if installed (it can cause issues with the PETSC build)
```

```
# example code to remove anaconda3 from path
# export PATH=$(echo "$PATH" | sed -e 's/:\/Users\/username\/anaconda3\/bin//')
# export PATH=$(echo "$PATH" | sed -e 's/\/Users\/username\/anaconda3\/bin://')

cd src
make -j8
```

———- End build ———

———- BEGIN Makefile ———

```
ALL: TDSE

include ${SLEPC_DIR}/lib/slepc/conf/slepc_rules
include ${SLEPC_DIR}/lib/slepc/conf/slepc_variables

CFLAGS = ${PETSC_CC_INCLUDES}
CXXFLAGS= ${PETSC_CXX_INCLUDES} -std=c++14 -Wall
FFLAGS = ${PETSC_FC_INCLUDES}
INCLUDES = -lhdf5_hl_cpp -lhdf5_cpp -lgsl -lgslcblas -lboost_mpi ${SLEPC_EPS_LIB} -I/usr/local/include/
CXX = mpicxx

# put this on one line (the return is to allow it to fit on the page)
OBJECTS = TDSE.o Parameters.o ViewWrapper.o HDF5Wrapper.o Pulse.o PETSCWrapper.o Wavefunction.o
Hamiltonian.o Simulation.o Utils.o

TDSE: ${OBJECTS}
  ${CXX} -o TDSE ${OBJECTS} ${PETSC_LIB} ${INCLUDES}
  cp TDSE ../bin
```

———- End Makefile ———

### 2.7 bashrc

To remove the need to set environmental variables every time you log in, I recommend setting them in your ${HOME}/.bashrc (or ${HOME}/.zshrc if using the ZSH shell). Here are the following lines of code to add:
"export PETSC_DIR=${PATH_TO_PETSC_INSTALL}"
"export SLEPC_DIR=${PATH_TO_SLEPC_INSTALL}"
(without the quotes and replacing ${PATH_TO_PETSC_INSTALL} and ${PATH_TO_SLEPC_INSTALL} as done before)

## 3 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 ($\hat{H}$) 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), the dynamics are non relativistic (the electron energy is much less than its rest mass), 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 $\nabla_e^2$ means the derivatives only act on the $e$th 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 3.4.

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 3.3.2) to avoid the singularity at $r_{n,e} = 0$ though this is not required for full 3D simulations. Some reviewers are not a fan of soft cores, so be careful when deciding to use them. 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 3.3.9.

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 use a single (many) active electron (SAE) potential. In that case, your Hamiltonian becomes

$$\hat{H} = \frac{\hat{\mathbf{p}}^2}{2} - \frac{\mathbf{A}(t)\cdot\hat{\mathbf{p}}}{c} + V(r) \tag{7}$$

for single active electron and

$$\hat{H} = \sum_e \left(\frac{\hat{\mathbf{p}}_e^2}{2} - \frac{\mathbf{A}(t)\cdot\hat{\mathbf{p}}_e}{c}\right) + \sum_{e_1<e_2} \frac{1}{r_{e_1,e_2}} + V(r) \tag{8}$$

for more than one electron where $V(r)$ is the single (many) active electron potential. See Sec 3.3.9 and Sec 3.3.10 for a short discussion on these potentials.

## 3.1 Spacial Derivatives

For spacial derivatives, this code utilized finite differences (though other methods are being implemented). 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 $\psi$ known at various evenly spaced grid points labeled by $n$ as

$$\frac{d^2}{dx^2}\psi_n = \frac{1}{\Delta x^2}\sum_i c_i\psi_i \tag{9}$$

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 9 we get

$$\frac{d^2}{dx^2}\psi_n = \frac{1}{\Delta x^2}\left(-\psi_{n-1} + 2\psi_n - \psi_{n+1}\right) \tag{10}$$

Now we can write this for the point $\psi_{n+1}$ giving us

$$\frac{d^2}{dx^2}\psi_{n+1} = \frac{1}{\Delta x^2}\left(-\psi_n + 2\psi_{n+1} - \psi_{n+2}\right) \tag{11}$$

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

$$\frac{d^2\psi}{dx^2} = \frac{1}{\Delta x^2}\begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix}\begin{bmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{N-1} \\ \psi_N \end{bmatrix} \tag{12}$$

One then uses a tensor product to produce the 2 and 3 dimensional versions.

## 3.2 Time Propagation

Time propagation is often performed using the split-operator method where the Hamiltonian ($\hat{H}$) split into its spatial dimensions, e.g. along ($z$) and perpendicular ($\rho$) to the laser polarization direction. The resulting propagation scheme is

$$\psi(\mathbf{r}, t + \Delta t) \approx e^{-i\hat{H}_\rho\frac{\Delta t}{2}}e^{-i\hat{H}_z(t)\Delta t}e^{-i\hat{H}_\rho\frac{\Delta t}{2}}\psi(\mathbf{r}, t). \tag{13}$$

where

$$e^{-i\hat{H}\Delta t} \approx \frac{1 - i\frac{\Delta t}{2}\hat{H}}{1 + i\frac{\Delta t}{2}\hat{H}} \tag{14}$$

produces a set of tridiagonal matrices which can be solved with $\mathcal{O}(\mathcal{N})$ operations and $\mathcal{O}(\mathcal{N})$ memory. However, parallelization of such a method on a modern supercomputer with distributed memory can be cumbersome, requiring multiple all-to-all Message Passing Interface (MPI) messages during each time step.

Instead, we avoid splitting the Hamiltonian and propagate the total Hamiltonian in time using a second order Crank-Nicolson scheme where

$$\psi(\mathbf{r}, t + \Delta t) \approx e^{-i\hat{H}\Delta t}\psi(\mathbf{r}, t). \tag{15}$$

We note that the Crank-Nicolson method is not tridiagonal (due to a tensor product) and a direct solution would require $\mathcal{O}(\mathcal{N}^3)$ operations and $\mathcal{O}(\mathcal{N}^2)$ memory which is significantly more than in the split operator method. However, the system of equations in the full Crank-Nicolson method is sparse and iterative methods can be used to vastly accelerate the time propagation. We utilize the Generalized Minimal Residual Method (GMRES), implemented in PETSc, which solves the sparse system of linear equations in $\mathcal{O}(\mathcal{N}\log(\mathcal{N}))$ operations and $\mathcal{O}(\mathcal{N})$ memory. The PETSc library makes it straightforward to parallelize the Crank-Nicolson method on modern supercomputers with distributed memory. On a local supercomputer (Summit, CU Boulder), we achieved super-linear scaling up to 3,000+ cores allowing us to complete simulations in a matter of hours that would take weeks running on a high-end workstation.

#### 3.2.1 Exterior Complex Scaling (ECS)

To absorb any outgoing wave packets, we utilize an exterior complex scaling (ECS). This is equivalent to changing the spacial step used in finite difference from $x_{n+1} = x_n + dx$ to $x_{n+1} = x_n + e^{i\eta}dx$ where $\eta = \pi/4.0$ seams to work well. This leads to an exponential decay of the wavefunction in the absorbing region.

### 3.3 Atoms and Molecules

The code allows for nuclei to be placed at any location assuming the coordinate system supports it. This allows for molecular targets to be build by hand with each nuclei having a unique potential. The potentials available are laid out below. If you need to add your own potential please follow the step by step procedure in Sec 3.3.11. **Please don't hard code potentials!**

#### 3.3.1 Molecular Targets

To model molecular targets, you can place the center of each nuclei at various locations on the grid. You can then add coulomb, Gaussian, exponential, and Yukawa potentials to approximate the potential made by that nuclei. By including many nuclei, you can simulate complicated molecular targets with a one or two active electron potential.

#### 3.3.2 Soft Core

A soft core can be used to calculate distances. This removes the singularity of the coulomb potential, however, reviewers are often not a fan of such a method. Use a soft core only when required. The soft core is calculated using

$$r_{soft} = \sqrt{r^2 + \alpha^2} \tag{16}$$

where $r$ is the euclidean distance $r = \sqrt{\sum_i x_i^2}$ and $\alpha$ is the soft core parameter.

#### 3.3.3 Hydrogen Like (coulomb) Potentials

The code provides a coulomb potential with the form

$$V(r) = -\frac{z}{r_{soft}} \tag{17}$$

where $z$ is the nuclear charge and $r$ is the euclidean distance if $\alpha = 0$.

#### 3.3.4 Donut Potentials

For various projects, it has been useful to add a donut potential to model a molecule like $C_{60}$ surrounding an atom. This can be done by adjusting the $r_0$ parameter of the potential being used to the radius of the molecule. Setting $r_0 = 0$ provides the usual radial potentials used in atomic single active electron potentials.

### 3.3.5 Gaussian Potentials

Gaussian potentials take the form:

$$V(r) = -\sum_i a_i e^{-\frac{1}{2}(c_i(r-r_{0,i}))^2} \tag{18}$$

where $a_i$ is the amplitude, $c_i$ is the decay rate, $r$ is the Euclidean distance, and $r_{0,i}$ is the radius of the donut (Sec 3.3.4).

### 3.3.6 Exponential Potentials

Exponential potentials take the form:

$$V(r) = -\sum_i a_i e^{-c_i|r-r_{0,i}|} \tag{19}$$

where $a_i$ is the amplitude, $c_i$ is the decay rate, $r$ is the Euclidean distance, and $r_{0,i}$ is the radius of the donut (Sec 3.3.4).

### 3.3.7 Square Well Potentials

Gaussian potentials take the form:

$$V(r) = \sum_i \begin{cases} -a_i, & r_{0,i} \leq r \leq r_{0,i} + \Delta_i \\ 0, & \text{otherwise} \end{cases} \tag{20}$$

where $a_i$ is the amplitude, $\Delta_i$ is the width, $r$ is the Euclidean distance, and $r_{0,i}$ is the radius of the donut (Sec 3.3.4).

### 3.3.8 Yukawa Potentials

Yukawa potentials take the form:

$$V(r) = -\sum_i \frac{a_i e^{-c_i|r-r_{0,i}|}}{|r_{soft} - r_{0,i}|} \tag{21}$$

where $a_i$ is the amplitude, $c_i$ is the decay rate, $r$ is the Euclidean distance, $r_{soft}$ is the soft core distance (Sec 3.3.2), and $r_{0,i}$ is the radius of the donut (Sec 3.3.4).

### 3.3.9 Single Active Electron

Bryn's Single active electron (SAE) potentials for atomic targets allow for a multi electron system, such as Helium, to be approximated by studying on active electron. His potentials take the form:

$$V(r, r_{soft}) = -\frac{z}{r_{soft}} - \frac{Z_c e^{-cr}}{r_{soft}} - \sum_n a_n e^{-b_n r} \tag{22}$$

where $r_{soft}$ is the soft core distance from the nuclei's location and $r$ is the euclidean distance form the nuclei's location. Talk to Brynn or I to get parameters for different targets. There exists many other SAE potentials and the code now supports most forms.

### 3.3.10 Two Active Electrons

This code is slowly gaining support for two active electron simulations. We have interest in developing two active electron (TAE) potentials, though little work has been done on this front.

#### 3.3.11 Adding New Potentials

**Please don't hard code potentials!**

Adding potentials that are not currently available only takes a 10ish minutes. Providing access from the input file allows all users of the code to easily use various potentials. Hard coding potentials is a poor practice and leads to unmaintainable and difficult to use code. **Please don't hard code potentials!**

To add a new potential follow these steps and make a pull request.

- Start new branch
- Add the parameters to the input.json file under "target - nuclei" in the same format as other potentials
- Add new double arrays for the potential parameters in Parameters.h
  - See "exponential_r_0" to see an example
  - Don't forget to add something like "exponential_size"
- Read in the parameter in Parameters.cpp
  - Allocate the memory
  - Read in parameters
  - Clean up memory under $\tilde{\text{P}}$arameters
  - Create getter functions to allow access to the pointer
  - Follow "exponential_r_0" for an example
- Write the data to disc in the HDF5Wrapper.cpp file.
  - Goes in the WriteHeader function
- Add pointers to Hamiltonian.h file
- Construct potential in Hamiltonian.cpp file
  - Read in potential values in constructor
  - Add your potential form to the GetNucleiTerm functions (**there are two of them**)
- compile your code
- test potential
- make pull request

### 3.4 Lasers

To ensure a laser is "physical", it is required that the electric field ($E(t)$) integrates to zero. The easiest way to handle this is to set the vector potential directly ($A(t)$) and calculate $E(t)$ such that

$$A(t) = \frac{cE_0}{\omega_A} f(t) \sin(\omega_A(t-\tau_0) + \phi_A) \tag{23}$$

and

$$\begin{aligned} E(t) =& -\frac{1}{c}\frac{\partial}{\partial t} A(t) \\ =& -E_0 f(t) \cos(\omega_A(t-\tau_0) + \phi_A) \\ & -\frac{E_0}{\omega_A}\frac{\partial f(t)}{\partial t} \sin(\omega_A(t-\tau_0) + \phi_A). \end{aligned} \tag{24}$$

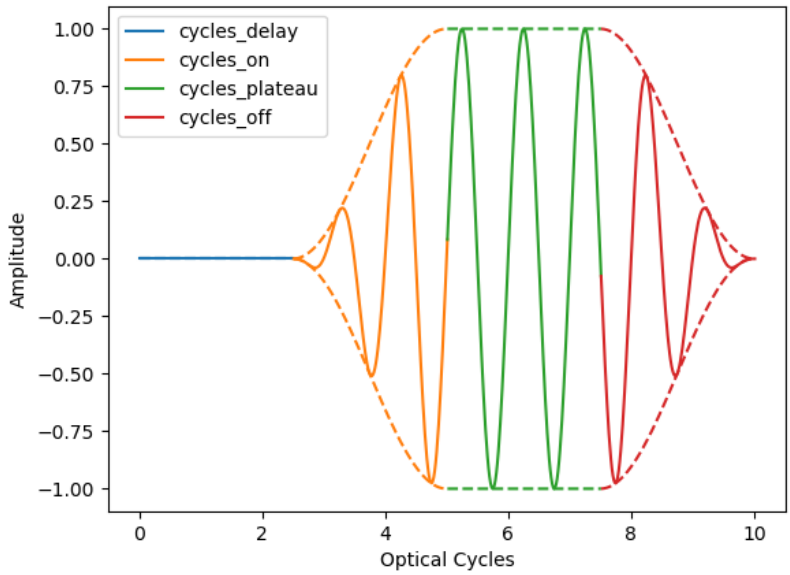


Figure 1: A graphical depiction of the various laser parameters described in Sec. 8.3. The dotted lines show the laser envelope and the solid lines are the vector potential. The various colors show what regions are effected by the various laser parameters. Depicted is a $\sin^2$ pulse.

$f(t)$, $\omega_A$, $c$, $E_0$, $\tau_0$, and $\phi_A$ are the envelope function, central frequency of the vector potential, speed of light, amplitude of the electric field, the time the carrier envelope phase is set, and carrier envelope field (CEP) of the vector potential respectively. The central frequency of the electric field $\omega_E$ is the important one. To set is one must correct for it in $\omega_A$ using the following relation

$$\frac{\omega_E}{\omega_A} = \frac{1 + \sqrt{1 + \mu/N^2}}{2} \tag{25}$$

with N being the number of cycles in $\tau$ of the envelope. For the sine squared

$$f(t) = \sin^2\left(\frac{\pi t}{\tau}\right) \tag{26}$$

and Gaussian

$$f(t) = \exp\left(-\ln(2)\left(\frac{2(t-\tau_0)}{\tau}\right)^2\right) \tag{27}$$

envelopes used in this code, the parameter $\mu$ for them are

$$\mu_{sin^2} = 4\arcsin(e^{-1/4})^2 \tag{28}$$

and

$$\mu_{gauss} = \frac{8\ln(2.0)}{\pi^2}. \tag{29}$$

**These values are only valid if the cycles_on and cycles_off parameters are the same and cycles_plateau = 0.0.**

The code itself supports slightly more general pulse shapes than the standard sine squared or Gaussian pulse. The code allows for the pulse to be delayed before starting using the cycles_delay parameter, ramped on for a period of time using the cycles_on parameter, then held at its max intensity for using cycles_plateau, and ramped off using the cycles_off parameter. See Fig. 1 for a graphical representation of the parameters.

This allows for the creation of most pulses. The code also supports multiple pulses that can be controlled independently of each other. The on and off ramps can either be a $\sin^n$ or Gaussian shape. You can mixed powers of sine (**must be even**) in a single pulse, however, if Gaussian is chosen both ramps are Gaussian though they can half different widths. For the Gaussian pulses, the code propagates for 5 times the full width half max (FWHM) on the ramp on and off. This brings the Gaussian close to zero since a Gaussian is never truly zero. This may want to be tested further at some point, though it will have a minimal impact on the physics and a significant impact on runtime if increased/decreased.

Using flat top pulses (cycles_plateau $> 0$) can clean up many results, however, they are extremely difficult to produce experimentally for pulses less than a picosecond. Due to this, reviewers may have issues with these pulses. Just something to be aware of.

#### 3.4.1 Experiment type

TODO

- "default"
- "File" describe input file
- "streaking" 2 pulses, tau_delay

# 4 Running Calculations

## 4.1 Running the Code

### 4.1.1 Basic usage

Once the code has been properly built, a binary file "${TDSE_ROOT}/bin/TDSE" should exist with ${TDSE_ROOT} as the path to the TDSE repository. You will need to define this variable yourself, or type the path out. To run the code, change directory (cd) to a directory with "input.json" file that contains the parameters for the simulation you wish to run. (see Sec. 5 for details on the "input.json" file) Then execute the binary file. This can be done by just typing

```
${TDSE_ROOT}/bin/TDSE
```

into the command line (replacing ${TDSE_ROOT}) and pressing enter.

An example output is on the following page (Sec 4.1.2).

### 4.1.2 Example output

Not all parts of the output will exist depending on the options you have chosen in the input file. The output file is generally easy to understand. If you have questions let me know.

```
****************** Setting up Simulation ******************

Simulation running on 1 Processors
Reading input file: input.json
Validating input
Input valid
Reading input complete
Creating pulses
Pulses created
Creating Wavefunction
Checkpointing Psi: 0
Wavefunction created
Creating Hamiltonian
Hamiltonian Created
Creating Simulation
Simulation Created

****************** Eigen State Calculation ******************


Calculating the lowest 1 eigenvectors using SLEPC
Eigen (-0.904578,-1.94079e-14) (0,0) 1
Checkpointing Wavefunction in He.h5:  0

*********************** Propagation ************************


Propagating in time
Total writes: 2
Starting propagation
Checkpointing Psi: 1

Iteration: 2000
Pulse ends: 2940
Average time for time-step: 1.77576
Checkpointing Psi: 2
Checkpoint time: 0.199936
Checkpointing Psi: 3
Checkpointing Psi Projections: 1

Propagating until step: 2940
Checkpointing Psi: 4
Checkpointing Psi Projections: 2

******************** Simulation Complete *********************

Deleting Hamiltonian
Deleting Wavefunction
Deleting Pulse
```

### 4.1.3 Command line options

Here are a few command line options to make this codes log file a bit more useful. Running the code with command line options will look like

```
${TDSE_ROOT}/bin/TDSE -command_line_option_0 -command_line_option_1
```

If you are using SLEPc I recommend adding

```
-eps_monitor
```

to your command line. This will allow you to monitor the convergence of your eigen state calculations
For profiling the code use

```
-log_view
```

This will provide a large amount of details relating to how the code preformed during a run. It is useful for estimating future job runtimes and profiling the overall code.
If you wish to change to propagation solver (default is GMRES) use

```
-prop_ksp_type ${KSP_TYPE}
```

See PETSc's documentation for the available options. Use "-log_view" to compare them.
If you want to change the solver for the eigen state method (only for Power method) use

```
-eigen_ksp_type ${KSP_TYPE}
```

A recommended setup if you keep getting divergence issues is

```
-eigen_ksp_type preonly -eigen_pc_type lu -eigen_pc_factor_mat_solver_package superlu_dist
```

### 4.1.4 Parallel Execution

This code scales well on modern HPC systems. To utilize this, you can run the code in parrallel utilizing "mpiexec" or a similar command. The bash command to run the code will look like

```
mpiexec -n ${number_of_processors} {TDSE_ROOT}/bin/TDSE -command_line_options
```

where ${number_of_processors} is the number of processors you wish to run this code on. We have used upwards of 3,000 without issue. The number of processors being used is printed near the top of the output.

### 4.1.5 Ground State Calculations

If you wish to produce only ground states that can be read in to other calculations, set "propagation" to 0 (Sec 5.2.35) and "state_solver" to something other than "File" (Sec. 5.2.43). This will produce a ground state file once the calculation completes. This can be read in using the "name" parameter (Sec 5.2.45). By setting the full path in name (leaving off the .h5), many calculations can read the same file. This saves storage space and computational time.

### 4.1.6 Time Propagation

To propagate in the laser field, set "propagation" to 1 (Sec 5.2.35) and ensure your laser (Sec 5.2.15) is set up correctly. It is also useful to set "state_solver" to "File" (Sec. 5.2.43) if you have already calculated a set of ground states.

### 4.1.7 Restart mode

This code provides the ability to restart a simulation that has timed out or crashed during laser propagation. To run a new simulation that starts at the beginning of the laser pulse set "restart" to 0 (Sec 5.2.36). If you wish to start from the last checkpoint in the TDSE.h5 file set "restart" to 1.

#### 4.1.8 Daisey Chain Script

TODO

#### 4.1.9 Visualization of Results

TODO

#### 4.1.10 Create Observables file

The output from this code is stored in very few HDF5 files. This keeps the run directory clean and avoids running into file count limits imposed at some super computing facilities. However, large files are hard to transfer, and often the wavefunctions are not needed to analyze the physics. To produce a smaller data file with many useful obserables, go to the directory your simulation was run in. The execute the "make_obs.sh" which can be found at "${TDSE_ROOT}/scripts/make_obs.sh". If the base script doesn't satisfy your needs, feel free to edit away as it is a relatively easy to understand script.

### 4.2 TDSE Best Practices

#### 4.2.1 Sharing Ground State files

TODO

#### 4.2.2 Convergence Tests

TODO

#### 4.2.3 Minimizing IO

Writing wavefunctions and other large vectors to disk is a time consuming task (can take many minutes) and the data produced can add up quickly (I have seen upwards of a terrabyte for one simulation). Because of this, it is best to set "write_frequency_checkpoint" to a large number if the wavefunction itself is not needed. However, if the number is set to high, no checkpoints to restart will be available in case a computer crashes or a simulation times out. It is up to you to determining the best frequency on your particular situation, though 1-4 hours between checkpoints is a good starting value on modern super computers.

#### 4.2.4 Vector Potential vs Electric Field

TODO

#### 4.2.5 Flat Top Pulse

Utilizing flat top pulses (cycles_plateau ¿ 0) can make various observables cleaner. However, flat top pulses with durations less than a picosecond are difficult to create in experiment. As a result, there uses should be limited to understanding and highlighting physical phenomena that can be observed without the need for a flat top pulse.

#### 4.2.6 Soft Core Potentials

Soft cores are a common way of adjusting ground state energies to match experimental or theoretical values. They also remove the issues with the coulomb singularity. However, some reviewers have been known to have issues with them. If it is possible to avoid, it is best not to use them.

#### 4.2.7 Extracting Photoelectron Spectra

TODO

#### 4.2.8 High Harmonic Generation

TODO

#### 4.2.9 Bound State Populations

TODO

# 5 Input

For the code, the only thing that needs to be in the directory is the "input.json" file. This will describe the laser, atomic/molecular target, and various other things required for the code to run. This section will go through what each parameter means and a list of options for each parameter when applicable.

The input files are in the json format. This formate is standardized, however, missing or extra commas, brackets, braces, ect. can be tricky to track down if you are not familiar with the json format. **Please take the time to learn json now. You will thank yourself later.**

**Make sure to use ASCII characters, copy and paste could cause issues!**

## 5.1 Example File

The following is an input file for simulating an 8 cycle, 800nm, sine squared pulse incident on atomic Hydrogen. This will produce a well converged High Harmonic Spectrum, though it may take a few hours to run. For a faster simulation that will be close to right, you can increase "delta_x_max" and "delta_x_min" to a larger number (say 0.5). You will need to do this for both dimensions.

**Make sure to use ASCII characters, copy and paste could cause issues!**

(current versions of example files can be found in the "example" directory)

```
{
  "alpha": 0.0,
  "coordinate_system": "Cylindrical",
  "delta_t": 0.1,
  "dimensions": [
  {
      "delta_x_max": 0.1,
      "delta_x_max_start": 4.0,
      "delta_x_min": 0.1,
      "delta_x_min_end": 4.0,
      "dim_size": 100.0
    },
    {
      "delta_x_max": 0.1,
      "delta_x_max_start": 4.0,
      "delta_x_min": 0.1,
      "delta_x_min_end": 4.0,
      "dim_size": 200.0
    }
  ],
  "ee_soft_core": 0.01,
  "field_max_states": 0,
  "free_propagate": 0,
  "gauge": "Velocity",
  "gobbler": 0.9,
  "laser": {
    "experiment_type": "default",
    "pulses": [
```

```
    {
      "cep": 0.0,
      "cycles_delay": 0.0,
      "cycles_off": 4.0,
      "cycles_on": 4.0,
      "cycles_plateau": 0.0,
      "ellipticity": 0.0,
      "energy": 0.057,
      "gaussian_length": 5.0,
      "helicity": "left",
      "intensity": 1e14,
      "polarization_vector": [0.0, 1.0, 0.0],
      "power_off": 2.0,
      "power_on": 2.0,
      "poynting_vector": [0.0, 0.0, 1.0],
      "pulse_shape": "sin"
    }
  ]
},
"num_electrons": 1,
"order": 2,
"propagate": 1,
"restart": 0,
"sigma": 3.0,
"start_state": {
  "amplitude": [1.0],
  "index": [0],
  "phase": [0.0]
},
"state_solver": "SLEPC",
"states": 5,
"target": {
  "name": "H",
  "nuclei": [
    {
     "exponential_r_0": [0.0],
     "exponential_amplitude": [0.0],
     "exponential_decay_rate": [0.0],
     "gaussian_r_0": [0.0],
     "gaussian_amplitude": [0.0],
     "gaussian_decay_rate": [0.0],
     "location": [0.0, 0.0, 0.0],
     "square_well_r_0": [0.0],
     "square_well_amplitude": [0.0],
     "square_well_width": [0.0],
     "yukawa_r_0": [0.0],
     "yukawa_amplitude": [0.0],
     "yukawa_decay_rate": [0.0],
     "z": 1.0
    }
  ]
},
"tol": 1e-10,
"write_frequency_checkpoint": 1000,
```

```
  "write_frequency_eigin_state": 1000,
  "write_frequency_observables": 1
}
```

## 5.2 Parameters

The following is my attempt as describing what each parameter does. Each parameter is a subsubsection and they should come in the order of the example input file in Sec 5.1. Sub section names for nested parameters include a "-" to show that the parameter is either part of a list or dictionary of a different parameter. This is shown in Sec 5.2.9 as "dim_size" is a sub parameter of the "dimensions" parameter. Some of the parameters have extra descriptions that are critical to reproduce results from other codes. This may lead to some lengthy discussion that may not be useful for all users. If you have questions on particular parameters or more general questions, let me know.

**Make sure to use ASCII characters, copy and paste could cause issues!**

### 5.2.1 alpha

alpha is the soft core parameter described in Sec 3.3.2 used in nuclear interactions. A floating point number can be input, and a value of 0.0 leads to the standard Euclidean distance $r = \sqrt{x^2 + y^2 + \cdots}$. It is used in the Coulomb (Hydrogen like) potentials. It is also used for any $r$ that comes in the denominator of the Single Active Electron (SAE) Potentials in Sec 3.3.9. Any $r$ that comes in the numerator is the standard Euclidean distance. The soft core takes the mathematical form of

$$r_{soft} = \sqrt{r^2 + \alpha^2} \tag{30}$$

where $r$ is the standard Euclidean distance.

### 5.2.2 coordinate_system

This describes the available coordinate systems available. Current list of available coordinate systems is (make sure to use ASCII quotes, don't just copy and paste)

- "Cartesian"
- "Cylindrical"
- "RBF" (beta)

The "Cartesian" coordinate system supports 1-3 dimensions with up to 2 electrons with any laser polarization that can be described in that number of dimensions. Though 2 electron calculations are computationally limiting. "Cylindrical" is a discretization in cylindrical coordinates for linear polarization on atoms or aligned linear molecules. It is 2 dimensional ($\rho$ and $z$) and runs much faster than a 3D simulation would. The "RBF" code is very much in beta testing and requires and external package. Only those wishing to get their hands dirty developing the method further should use this code. Talk to me if this is you.

### 5.2.3 delta_t

delta_t is the time step used in the simulation in atomic units (floating point number). Time steps in the range of 0.05 to 0.1 are typically sufficient for a simulation, however, one must run a convergence test non the particular observable you're interested in.

### 5.2.4 dimensions

dimensions is a list of json objects (also known as dictionaries). Each dictionary provides the size of a new dimension. In the Cartesian coordinate system, the dimensions are added in the order x, y, z. If only 2 dictionaries are included, a 2D simulation is preformed. For Cylindrical coordinates the dimensions are added in the order $\rho$ and $z$.

**Note**: Each dimension can have a minimum grid spacing and a maximum grid spacing with a sine squared ramp connecting them. This appears to work for ground state calculations, however, time propagation can have issues. Take extra care if you wish to use this feature. If "delta_x_min" = "delta_x_max" are the same value, the code has been well tested and will produce good results (you just have to get past the error checks by moving "delta_x_max_start" and "delta_x_min_end" away from the boundaries of your grid).

#### 5.2.5 dimensions - delta_x_max

delta_x_max is the max grid spacing in atomic units (outside edge of the grid). This is useful to describe highly excited states and outgoing wave packets, though it is not 100% stable. Set equal to delta_x_min to use the well tested code. Typical values for converged calculations on a uniform grid are less than or equal to 0.1. See the note in Sec 5.2.4 for more details on nonuniform grid.

#### 5.2.6 dimensions - delta_x_max_start

delta_x_max_start is the distance from the origin that the delta_x_max is used to increase the grid size. See the note in Sec 5.2.4 for more details on nonuniform grid.

#### 5.2.7 dimensions - delta_x_min

delta_x_min is the min grid spacing in atomic units (outside edge of the grid). This is useful to describe highly excited states and outgoing wave packets, though it is not 100% stable. Set equal to delta_x_max to use the well tested code. Typical values for converged calculations on a uniform grid are less than or equal to 0.1. See the note in Sec 5.2.4 for more details on nonuniform grid.

#### 5.2.8 dimensions - delta_x_min_end

delta_x_min_end is the distance from the origin that the sine squared ramp up between delta_x_min and delta_x_max begins. See the note in Sec 5.2.4 for more details on nonuniform grid.

#### 5.2.9 dimensions - dim_size

dim_size is the total size of the dimension in atomic units. For all Cartesian and the $z$ axis in Cylindrical, the grid will go from -dim_size/2 to dim_size/2. For the $\rho$ dimension in Cylindrical, the grid will go from 0 to dim_size.

#### 5.2.10 field_max_states

field_max_states allows for the peak of the electric field or vector potential to be used for eigenstate calculations. Utilizing this parameter is still in beta. Testing is required if this parameter is not set to 0. The following values can be set

- "0" Field free (standard use case)
- "1" Field max (still in beta)

#### 5.2.11 ee_soft_core

ee_soft_core is the soft core parameter described in Sec 3.3.2 used in the electron-electron repulsion term.

#### 5.2.12 free_propagate

free_propagate is the number of time steps the wavefunction is propagated after the last laser "finishes" (integer). Total free propagation is delta_t * free_propagate in atomic units.

#### 5.2.13 gauge

gauge allows for calculation to be done in Velocity and Length gauge and runtime differences are negligible in the cases I've tested. Both should produce the same answer when fully converged. The options are

- "Velocity"
- "Length"

#### 5.2.14 gobbler

This sets the size of the exterior complex scaling (ECS) potential (Sec 3.2.1) which absorbs outgoing wavepackets. A value of 0.9 means that 90% of the grid is normal and the outer 5% on each side contains an ECS potential.

#### 5.2.15 laser

The laser parameter is a dictionary that contains the particular laser being used in the calculation.

#### 5.2.16 laser - experiment_type

The experiment_type is how the laser is being used. The options are

- "default"
- "File"
- "streaking"

Below is documentation for "default". Details on the others can be found in Sec 3.4.1.

#### 5.2.17 laser - pulses

pulses is a list of dictionaries used to describe a pulse. Each component inside a single pulse is in the units of that pulse such as the length of a single cycle is based on the frequency of that particular pulse.

#### 5.2.18 laser - pulses - cep

cep is the carrier envelope phase (CEP) which is $\phi_A$ in Eq. 23. It is in units of $2\pi$, so a value of 0.5 gives $\phi_A = \pi$. The CEP is defined off of $t_0$ which is defined at the end of cycles_on (peak of laser field/beginning of the plateau).

#### 5.2.19 laser - pulses - cycles_delay

cycles_delay is the number of cycles until the ramp up of the pulse begins. See Fig. 1 for a graphical representation.

#### 5.2.20 laser - pulses - cycles_off

cycles_off is the number of cycles that ramps the pulse back to zero. See Fig. 1 for a graphical representation. Note that this is the full size for "sin" envelopes and the FWHM for "Gaussian". The "Gaussian" shapes are propagated for 5 times the cycles_on to allow for the envelope to approach zero.

#### 5.2.21 laser - pulses - cycles_on

cycles_on is the number of cycles that ramps the pulse from zero to its max value. See Fig. 1 for a graphical representation. Note that this is the full size for "sin" envelopes and the FWHM for "Gaussian". The "Gaussian" shapes are propagated for 5 times the cycles_on to allow for the envelope to approach zero.

#### 5.2.22 laser - pulses - cycles_plateau

cycles_plateau is the number of cycles the pulse remains at its max value. See Fig. 1 for a graphical representation.

Using flat top pulses (cycles_plateau $> 0$) can clean up many results, however, they are extremely difficult to produce experimentally for pulses less than a picosecond. Due to this, reviewers may have issues with these pulses. Just something to be aware of.

#### 5.2.23 laser - pulses - ellipticity

ellipticity is the ratio of the minor over major axis for the laser polarization. 0.0 gives linear polarization and 1.0 gives circular polarization.

#### 5.2.24 laser - pulses - energy

energy provides the central frequency ($\omega_A$) in Eq. 23. Given in atomic units (800nm $\approx$ 0.057a.u.). Note that the frequency shift (Eq. 25) described in Sec 3.4should be accounted for when pulses are around 15 optical cycles or less.

#### 5.2.25 laser - pulses - gaussian_length

gaussian_length is the number of times the FWHM a gaussian pulse is propagated for. The parameters cycles_on and cycles_off control the half width half max on the rising and falling parts of the pulse. This controls how close to zero the envelope gets to zero. For non gaussian pulse shapes, this is not read and set to one in the code. A good value appears to be 5.0 though feel free to test this out.

#### 5.2.26 laser - pulses - helicity

helicity is the handedness of elliptical and circular polarized lasers. It take the values of:

- "left"
- "right"

#### 5.2.27 laser - pulses - intensity

intensity is the peak intensity of the laser field. It is given in W/cm$^2$. Typical values are $10^{12}$ to $10^{15}$. Easiest to provide in the from $2e12$ for $2 * 10^{12}$, though it can be written out with all the zeros if one wishes.

#### 5.2.28 laser - pulses - polarization_vector

polarization_vector is a list that defines the direction of the major axis of the laser. It is normalized after being read in to avoid intensity scaling issues. The order of dimensions is the same as the coordinate system defined in Sec. 5.2.4.

#### 5.2.29 laser - pulses - power_off

power_off is the power a "sin" envelope function is raised to on the ramp off portion of the pulse scaled by cycles_off. The value must be an even power giving the envelope the form $\sin^{power\_off}$.

#### 5.2.30 laser - pulses - power_on

power_on is the power a "sin" envelope function is raised to on the ramp on portion of the pulse scaled by cycles_on. The value must be an even power giving the envelope the form $\sin^{power\_on}$.

#### 5.2.31 laser - pulses - poynting_vector

poynting_vector defines the direction of propagation of the laser. When combined with the polarization_vector and helicity, the coordinate system for the laser has been set. It is normalized after being read in to avoid intensity scaling issues. The order of dimensions is the same as the coordinate system defined in Sec. 5.2.4.

#### 5.2.32 laser - pulses - pulse_shape

pulse_shape gives the ramp functions for the pulse. They are either $\sin^n$ (where $n$ is set using power_on and power_off) or Gaussian. The values it takes are

- "sin"
- "gaussian"

#### 5.2.33 num_electrons

num_electrons is the number of electrons in the simulation. Having more than one electron should work but has not been throughly tested. Ask for 3 or more electrons for an easter egg.

#### 5.2.34 order

order is the order of finite difference used. Only 2nd order appears to work, however, and even order seams to work for ground state calculations. I recommend just leaving it set to 2.

#### 5.2.35 propagate

propagate controls if the wave function is propagated in the laser field. If set to 0, no propagation occurs. If set to 1, the wavefunction is propagated in the laser field.

#### 5.2.36 restart

restart allows the code to restart the propagation off the last checkpoint in the T. If set to 0, the code will start from the beginning of the pulse, and a fresh TDSE.h5 file will be produced. If set to 1, the code will restart off the last checkpoint in the TDSE.h5 file.

#### 5.2.37 sigma

sigma is the size of the Gaussian wavefunction used as an initial guess for the Power method. A value of 3.0 works well for all cases I have run across, but it could be adjusted to make the convergence go faster. I would recommend switching to SLEPC if you want things to speed up.

#### 5.2.38 states

states is the number of bound states of your potential you would like to calculate. It is also the number of states that are used to calculate populations in the code and projected out for removing the bound state population when calculating photoelectron spectra. This is a number, except when using the "Power" method as a state_solver. Then is it a list of dictionaries with target energies provided. Here is an example:

```
"states": [{"energy": -0.5}, {"energy": -0.125}]
```

#### 5.2.39 start_state

start_state is a dictionary that defines the initial conditions for time propagation. The initial state $\Psi$ is given by

$$\Psi = \sum_n a_n e^{i\phi_n} \psi_n^{idx} \tag{31}$$

$a_n$ is the amplitude, $psi_n$ is the bound state, $\phi_n$ is the phase, $n$ is the index into the "amplitude", "index", and "phase" arrays, and $idx$ is the index into the bound state data file given in the "index" array. The initial wavefunction is normalized to one after being read in, so the amplitude array's values are only relative amounts.

#### 5.2.40 start_state - amplitude

amplitude defines the $a_n$ in Eq. 31 where $n$ is the index into this array. The initial wavefunction is normalized to one after being read in, so the amplitude array's values are only relative amounts.

#### 5.2.41 start_state - index

index is the $idx$ wavefunction in the bound state file being used. It takes the value of $idx$ in Eq. 31 where $n$ is the index into this array. Note that the bound state file wavefunctions may have a random phase.

#### 5.2.42 start_state - phase

phase is the value of $\phi_n$ in Eq. 31 where $n$ is the index into this array. Note that the bound state file wavefunctions may have a random phase. If relative phase is important to your system take extra care when setting this.

#### 5.2.43 state_solver

state_solver it the method used to calculate ground states. "SLEPC" or "File" are the recommended modes of operation. All of the options are

- "File"
- "SLEPC"
- "Power"
- "ITP" (deprecated)

"File" will read from the file given in "target - name" with a ".h5" extension added to it. This allows for the same bound state file to be used by many calculations.

"SLEPC" utilized the SLEPc library to calculate eigen states. It uses the default SLEPc solver (currently Krylov Schur) though this can be changed by added the command line argument "-eps_type <method>" when running the code. See the SLEPc for more information on how to do this.

"Power" is the inverse shifted power method with target energies given by the "states" parameter. See "states" for an example on how to provide target energy values.

"ITP" is imaginary time propagation. This is no longer supported as much better methods have been implemented, though it does lead to an easter egg.

#### 5.2.44 target

target is how you define the potential (or target) in which you want to calculate bound states of or shine a laser on.

#### 5.2.45 target - name

name is the name of the bound state file. It can also be used to provide a path to place/read from the file in a different folder. You should avoid spaces and punctuation in file names (follow good file naming practices). Note that the ".h5" is added inside the code, so it should not appear in this name.

#### 5.2.46 target - nuclei

nuclei is a list that allows for nuclei to be placed at various places in space. Each can be coulomb or SAE (this may change later).

#### 5.2.47 target - nuclei - exponential_r_0

exponential_r_0 is the donut radius for an exponential potential. This is the value of $r_{0,i}$ in Eq. 19 with $i$ being the index into the array. Set to zero for atomic like behavior. See Sec 3.3.6 for exact form of the potential.

#### 5.2.48 target - nuclei - exponential_amplitude

exponential_amplitude is the amplitude for an exponential potential. This is the value of $a_i$ in Eq. 19 with $i$ being the index into the array. If set to 0 the potential has no effect. See Sec 3.3.6 for exact form of the potential.

#### 5.2.49 target - nuclei - exponential_decay_rate

exponential_decay_rate is the decay rate for an exponential potential. This is the value of $c_i$ in Eq. 19 with $i$ being the index into the array. See Sec 3.3.6 for exact form of the potential.

#### 5.2.50 target - nuclei - gaussian_r_0

gaussian_r_0 is the donut radius for a Gaussian potential. This is the value of $r_{0,i}$ in Eq. 18 with $i$ being the index into the array. Set to zero for atomic like behavior. See Sec 3.3.5 for exact form of the potential.

#### 5.2.51 target - nuclei - gaussian_amplitude

gaussian_amplitude is the amplitude for a Gaussian potential. This is the value of $a_i$ in Eq. 18 with $i$ being the index into the array. If set to 0 the potential has no effect. See Sec 3.3.5 for exact form of the potential.

#### 5.2.52 target - nuclei - gaussian_decay_rate

gaussian_decay_rate is the decay rate for a Gaussian potential. This is the value of $c_i$ in Eq. 18 with $i$ being the index into the array. See Sec 3.3.5 for exact form of the potential.

#### 5.2.53 target - nuclei - location

location is a list that corresponds to the nuclei center. The distances $r$ and $r_{soft}$ are calculated from it. The order of dimensions is the same as the coordinate system defined in Sec. 5.2.4.

#### 5.2.54 target - nuclei - square_well_r_0

square_well_r_0 is the donut radius for a square well potential. This is the value of $r_{0,i}$ in Eq. 20 with $i$ being the index into the array. Set to zero for atomic like behavior. See Sec 3.3.7 for exact form of the potential.

#### 5.2.55 target - nuclei - square_well_amplitude

square_well_amplitude is the amplitude or depth of a square well potential. This is the value of $a_i$ in Eq. 20 with $i$ being the index into the array. If set to 0 the potential has no effect. See Sec 3.3.7 for exact form of the potential.

#### 5.2.56 target - nuclei - square_well_width

square_well_width is the width of the square well potential. This is the value of $\Delta_i$ in Eq. 20 with $i$ being the index into the array. See Sec 3.3.7 for exact form of the potential.

#### 5.2.57 target - nuclei - yukawa_r_0

yukawa_r_0 is the donut radius for a Yukawa potential. This is the value of $r_{0,i}$ in Eq. 21 with $i$ being the index into the array. Set to zero for atomic like behavior. See Sec 3.3.8 for exact form of the potential.

#### 5.2.58 target - nuclei - yukawa_amplitude

yukawa_amplitude is the amplitude for a Yukawa potential. This is the value of $a_i$ in Eq. 21 with $i$ being the index into the array. If set to 0 the potential has no effect. See Sec 3.3.8 for exact form of the potential.

#### 5.2.59 target - nuclei - yukawa_decay_rate

yukawa_decay_rate is the decay rate for a Yukawa potential. This is the value of $c_i$ in Eq. 21 with $i$ being the index into the array. See Sec 3.3.8 for exact form of the potential.

#### 5.2.60 target - nuclei - z

z is the value of z in a coulomb like potential (Eq. 17). If set to 0 the potential has no effect. See Sec 3.3.3 for exact form of the potential.

#### 5.2.61 tol

tol is used whenever a convergence tolerance is needed in the code. The only place I remember is for the eigen state calculations. A value of $1e-10$ is a good starting point.

#### 5.2.62 write_frequency_checkpoint

write_frequency_checkpoint is the number of time steps between writing checkpoint files that allow the code to restart. Setting this to small can lead to massive data files. If it is to large, the code may crash before a checkpoint is written to disk. Typically you want a restart file written ever 20mins to a few hours of runtime depending on the size of the simulation. If you would like to make a video of the simulation, this can be set to a small value assuming you have the disk space.

#### 5.2.63 write_frequency_eigin_state

write_frequency_eigin_state is the number of iterations between making convergence/writing to standard out checks when using the power method. Add "-eps_monitor" to the run line in the bash script to monitor SLEPC calculations.

#### 5.2.64 write_frequency_observables

write_frequency_observables is the number of time steps between writing calculating observables such as dipole, dipole acceleration, norm, ecs population, ect. I recommend leaving this set to 1 as it is not that time consuming in the current version of the code.

# 6 Output

TODO

## 6.1 TDSE.h5

TODO

## 6.2 Pulse.h5

TODO

## 6.3 Target.h5

TODO

### 6.4 Observables.h5

TODO

# 7 Analysis

TODO

# 8 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.

### 8.1 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.

### 8.2 ViewWrapper

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

### 8.3 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 5. 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.

### 8.4 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

$$array[x + y * nx] = array[x][y] \tag{32}$$

and

$$array[x + y * nx + z * nx * ny] = array[x][y][z] \tag{33}$$

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.

### 8.5 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 5 on how to define pulses.

### 8.6 Hamiltonian

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

$$i\frac{\partial}{\partial t}\psi(x,t) = \hat{H}\psi(x,t) \tag{34}$$

Which leads to a Hamiltonian of

$$\hat{H} = \sum_e \left( \frac{\hat{\mathbf{p}}_e^2}{2} - \frac{\mathbf{A}(t)\cdot\hat{\mathbf{p}}_e}{c} - \sum_n \frac{Z_n}{r_{e,n}} \right) + \sum_{e_1<e_2} \frac{1}{r_{e_1,e_2}} \tag{35}$$

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.

$$\hat{H} = \frac{\hat{p_1}^2}{2} + \frac{\hat{p_2}^2}{2} - \frac{\vec{A}\cdot(\hat{p_1}+\hat{p_2})}{c} - \frac{Z}{x_1} - \frac{Z}{x_2} + \frac{1}{r_{12}} \tag{36}$$

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 3.

### 8.7 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.

## 9 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.

## 10 Wish list

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

## 11 Supporting works

### 11.1 Posters

TODO

### 11.2 Talks

TODO

### 11.3 Papers

TODO

## 12 Contributors

The following people have contributed to this code. No particular order is used.

- Joel Venzke
- Cory Goldsmith