

# Manual of QPC-TDSE

Zhao-Han Zhang <sup>\*†</sup>

July 12, 2023

---

<sup>\*</sup>*E-mail address:* zhangzhaohan@sjtu.edu.cn

<sup>†</sup>*affiliation:* Key Laboratory for Laser Plasmas (Ministry of Education) and School of Physics and Astronomy, Collaborative Innovation Center for IFSA (CICIFSA), Shanghai Jiao Tong University, Shanghai 200240, China

# Contents

<b>1</b>	<b>Overview</b>	<b>3</b>
<b>2</b>	<b>Installation</b>	<b>4</b>
2.1	Prerequisites	4
2.2	Environmental Settings	6
2.3	How-To-Use In Brief	7
2.4	Example Runs	8
2.5	Uninstall	9
<b>3</b>	<b>Implementation</b>	<b>10</b>
3.1	Parameter Interpretation	10
3.1.1	workflow parameters (1)	10
3.1.2	grid and basis parameters (2)	11
3.1.3	potential parameters (3)	13
3.1.4	wavefunction displayer parameters (4)	14
3.1.5	momentum distribution parameters (5)	14
3.1.6	eigenstate projection parameters (6)	16
3.1.7	propagation parameters (7)	16
3.1.8	observable parameters (8)	17
3.1.9	laser field parameters (9)	17
3.1.10	initial state parameters (10)	19
3.2	Library Structure	20
3.2.1	include/functions	20
3.2.2	include/utilities	20
3.2.3	include/libraries	20
3.2.4	include/resources	20
3.2.5	include/algorithm	20
3.2.6	include/structure	20
3.2.7	include/spherical	20

# 1 Overview

QPC-TDSE is a light-weighted C++ programme designed for the efficient numerical solution the full dimensional single-active-electron (SAE) time-dependent Schrödinger equation (TDSE) for a laser-driven atomic or fixed-nuclei molecular system.

The details of the algorithms and principles used to solve the equation, as well as the limitations on the types of physical problems it can handle, are discussed in the paper, which can be found at the [link](#).

A minimal introduction to running QPC-TDSE on your system can be found in section 2 of the guide. Section 3 provides a detailed description of how to set the proper parameters and explains the program's structure.

Even users with limited knowledge of C++ can still use the program by following the instructions in the guide. However, having a basic understanding of C++17 grammar is strongly recommended since the configuration files need to be edited accordingly. Additionally, familiarity with using a LINUX-type system and compiling/linking a C++ program using GNU make is required, as the code is specifically designed to run on such systems.

The license of QPC-TDSE is GPLv3. You can find a detailed description of the license on [this page](#). Please note that any use of the code is considered an agreement to abide by the terms of this license. While the GPLv3 grants numerous rights to the user, there are still some important points that need to be mentioned.

Just like most free software, QPC-TDSE does not guarantee to be free of bugs. The author of the code does not assume any legal responsibility for the results generated by the program. If you encounter any bugs or issues, or if you have constructive suggestions for improving the package, you are welcome to contact the author via email or by posting issues on the GitHub page.

It is important to note that running the program's binaries on your system without due care may result in unintended consequences. For instance, if you mistakenly use an incorrect name for the output, the program could overwrite existing files in an irretrievable manner, e.g. the results you obtained in previous runs. Therefore, it is strongly recommended to thoroughly review and examine your scripts before launching them. Furthermore, it should be mentioned that building the binaries using the "-B" flag in the make command could potentially overwrite files with the same name as the generated binaries. However, it is important to emphasize that all behaviors of the package are under your control. The program will only perform actions that you explicitly instruct it to do. Its sole interaction with your system is through creating or reading files. As long as you provide the correct paths and names when interacting with the program, there should be no risk or danger involved.

As an academic requirement, it is highly appreciated to appropriately cite this program when utilizing it in your projects or papers. By appropriately acknowledging and referencing this program, you not only contribute to the advancement of this project but also enrich the broader scientific community as a whole.

## 2 Installation

### 2.1 Prerequisites

You can compile the source code of QPC-TDSE on any hardware platform that supports AVX instruction sets, provided that the required libraries are installed correctly. However, please note that the provided instructions are tailored for GNU Make, utilizing a manually written makefile and two auxiliary files. These files contain compilation flags and environmental settings specifically designed to work with the GNU C/C++ compiler on Linux-based operating systems. We have not yet tested the compilation of QPC-TDSE with any other compilers. It is possible that the code may include compiler-oriented or compiler-specific elements (such as GCC-only pragma derivatives or array-like indexing of `_mm512d`). Therefore, using a different compiler for the current codebase could potentially lead to issues. It is recommended to stick with the GNU C/C++ compiler as outlined in the provided instructions to ensure compatibility and avoid any potential problems that may arise from using a different compiler.

Before using QPC please make sure that all of the necessary tools and libraries have already been installed and configured correctly on your system. Here are the list:

#### GNU Make (Make)

- (a) open source and available online
- (b) version 3.8 or higher

Make is usually pre-installed on most popular brands of linux distributions.

#### GNU Compiler Collections (GCC)

- (a) open source and available online
- (b) only the C/C++ compiler in it are used
- (c) version 8 or higher, since QPC-TDSE codes are written in C++17

GCC is usually pre-installed on most linux-type systems, however, some of them may not meet the version requirement.

#### GNU Scientific Libraries (GSL)

- (a) open source and available online
- (b) only the C APIs are called
- (c) version 2.6 or higher.

GSL is called for solving problems like Gauss-Legendre quadrature, numerical interpolation, spherical harmonics evaluation, Wigner-3j coefficients etc.

#### HDF5 Library (HDF)

- (a) open source and available online
- (b) only the C APIs are called
- (c) version 1.10.5 or higher.

HDF is called for saving/reading data into/from files, which serves as the only way to interact with file in QPC-TDSE currently. Most common analysis tools provides libraries to access hdf5 file. We provide example scripts using MATLAB.

## Intel one-API Base Toolkit

- (a) not open source but available online for free
- (b) the C APIs in Math Kernel Libraries (MKL) and the thread library libiomp5 are called
- (c) the latest version is recommended, version earlier than 2019 could cause unknown issues.

MKL is called for most of the standard dense/sparse linear algebra operations, fast fourier transforms, and fast evaluation of some special math functions. The author understands that linking to MKL with GCC is typically not the most popular way to use it.

Here is an example of how to install these tools and libraries on CentOS 7 (which can also be adapted for other Linux distributions). This assumes that you have a pre-installed GNU Make, GCC 4.8.5 and the necessary [build-essential prerequisites](#) for building higher version GCC are provided.

Please note that if you are performing a system-wide installation, you should run the commands as the root user. Otherwise, run them as a regular user. Suppose all tools and libraries are downloaded and build in **build\_dir** and then installed into **install\_dir**.

1. Create **build\_dir** and **install\_dir**. If your system has already been installed with a high version GCC, skip to step 4; Usually CentOS does not come with one, in this case go to next step.
2. If your system has already been installed with [GMP](#), [MPFR](#), [MPC](#) and [isl](#), skip to the next step. Otherwise download (probably tar.lz, tar.gz or tar.xz files) and move them to **build\_dir**. Extract them, for example by [GNU tar](#) (usually pre-installed):

```
tar -zxvf XXX.tar.gz
```

You will get files extracted into directories which contain the source codes to build the necessary libraries required by building GCC. The directories typically have the name **build\_dir/gmp-X.X.X**, **build\_dir/mpfr-X.X.X**, **build\_dir/mpc-X.X.X** and **build\_dir/isl-X.X** where X-s are version numbers. You should replace them with the correct ones in all commands and codes mentioned later. Install GMP, MPFR, MPC and isl by

```
mkdir install_dir/gmp-X.X.X
cd build_dir/gmp-X.X.X
./configure --prefix=install_dir/gmp-X.X.X
make && make check && make install
```

```
mkdir install_dir/mpfr-X.X.X
cd build_dir/mpfr-X.X.X
./configure --with-gmp=install_dir/gmp-X.X.X --prefix=install_dir/mpfr-X.X.X
make && make check && make install
```

```
mkdir install_dir/mpc-X.X.X
cd build_dir/mpc-X.X.X
./configure --with-gmp=install_dir/gmp-X.X.X --with-mpfr=install_dir/mpfr-4.1.1
--prefix=install_dir/mpfr-X.X.X
make && make check && make install
```

```
mkdir install_dir/isl-X.X
cd build_dir/isl-X.X
./configure --prefix=install_dir/isl-X.X --with-gmp-prefix=install_dir/gmp-X.X.X
--with-mpfr-prefix=install_dir/mpfr-X.X.X --with-mpc-prefix=install_dir/mpc-X.X.X
make && make install
```

```
mkdir install_dir/isl-X.X
cd build_dir/isl-X.X
./configure --prefix=install_dir/isl-X.X --with-gmp-prefix=install_dir/gmp-X.X.X
--with-mpfr-prefix=install_dir/mpfr-X.X.X --with-mpc-prefix=install_dir/mpc-X.X.X
make && make install
```

3. Download [GCC](#), move it to **install\_dir** and extract it. You will get **build\_dir/gcc-X.X.X**. Install GCC by

```
mkdir install_dir/gcc-X.X.X
cd build_dir/gcc-X.X.X
./configure --prefix=install_dir/gcc-X.X.X --with-gmp=install_dir/gmp-X.X.X
--with-mpfr=install_dir/mpfr-X.X.X --with-mpc=/usr/local/mpc-X.X.X
--with-isl=install_dir/isl-X.X --enable-shared --disable-multilib
--enable-threads=posix --enable-languages=c,c++ --with-long-double-128
make -j4 && make check && make install
```

If your GMP, MPFR, MPC or isl are not newly installed in a customized directory as in previous step, but pre-installed with your OS in a system directory (e.g. **/usr/lib64** or **/usr/local/lib64**), just remove the corresponding 'with' flags in the above commands. The configure script will automatically find them. Make sure these directories are visible in your system's search path when you do this. For example, by setting environmental variable **LD\_LIBRARY\_PATH** in your **~/.bash\_profile** and source it, or appending it to **/etc/ld.so.conf** and run **ldconfig**.

4. Download [GSL](#), move and extract in **build\_dir**. You will get **build\_dir/gsl-X.X.X**. Install it by

```
mkdir install_dir/gsl-X.X.X
cd build_dir/gsl-X.X.X
./configure --prefix=install_dir/gsl-X.X.X
make && make check && make install
```

5. Register a free account and download [HDF5](#), move and extract the compressed package. You will get **build\_dir/hdf5-X.X.X**. Be careful that building HDF5 and linking to HDF5 shared libraries require that **libdl.so**, **libz.so** and **libm.so** are visible in your system's search path. Install HDF5 by

```
mkdir install_dir/hdf5-X.X.X
cd build_dir/hdf5-X.X.X
./configure --prefix=install_dir/hdf5-X.X.X --enable-build-mode=production
--enable-cxx
make && make check && make install
```

6. Register a free account to get [Intel oneAPI Base Toolkit](#). Follow the instructions on the webpage to download and install it. We refer to the installation path of oneapi as **oneapi\_dir**. For default installation, it will typically be **/opt/intel/oneapi** for root users and **~/intel/oneapi** for non-root users. Basically QPC-TDSE only needs MKL and the thread library **libiomp5** in it.

At this stage, all prerequisites have been prepared. If your system has a package management softwares (like yum, apt etc.), the installation can be even simpler, although the above procedures may still work. However, installation using these procedures should be considered as a non-standard way because it will NOT inform the package manager about your dependency. To avoid problems it is recommended to install these libraries by the native approach on your system.

If you are user(s) of HPC Clusters or public platforms, you are usually not permitted to install libraries and softwares by yourself. Nevertheless, these tools/libraries are very popular ones and they have probably been installed. In that case, do ask your system manager about the exact path in which these tools/libraries are installed.

## 2.2 Environmental Settings

After having your system installed with the external libraries and compilation tools mentioned in sec.2.1, you may follow the instruction in this subsection to install QPC-TDSE.

1. Extract the compressed file.

Suppose the extracted files are put into a directory, say **workpath**. This will be your workpath of conducting all simulations with QPC-TDSE.

2. Prepare enviromental variables.

There are two files in **workpath/env**, **workpath/env/make\_common.mk** and **workpath/env/path.mk**. Both of them are included in **workpath/makefile**. Set enviromental variables which specifies the absolute path of external libraries in **workpath/env/path.mk**. If you install the external libraries according to the example in 2.1, they should be

```

QPC_PATH?=workpath
GCC_PATH?=install_dir/gcc-X.X.X
HDF_PATH?=install_dir/hdf5-X.X.X
GSL_PATH?=install_dir/gsl-X.X.X
MKL_PATH?=oneapi_dir/mkl/latest
ICX_PATH?=oneapi_dir/compiler/latest/linux

```

Leave **QPC\_PATH/env/make.common.mk** untouched unless you need to modify the compilation flags.

3. Finish the install.

## 2.3 How-To-Use In Brief

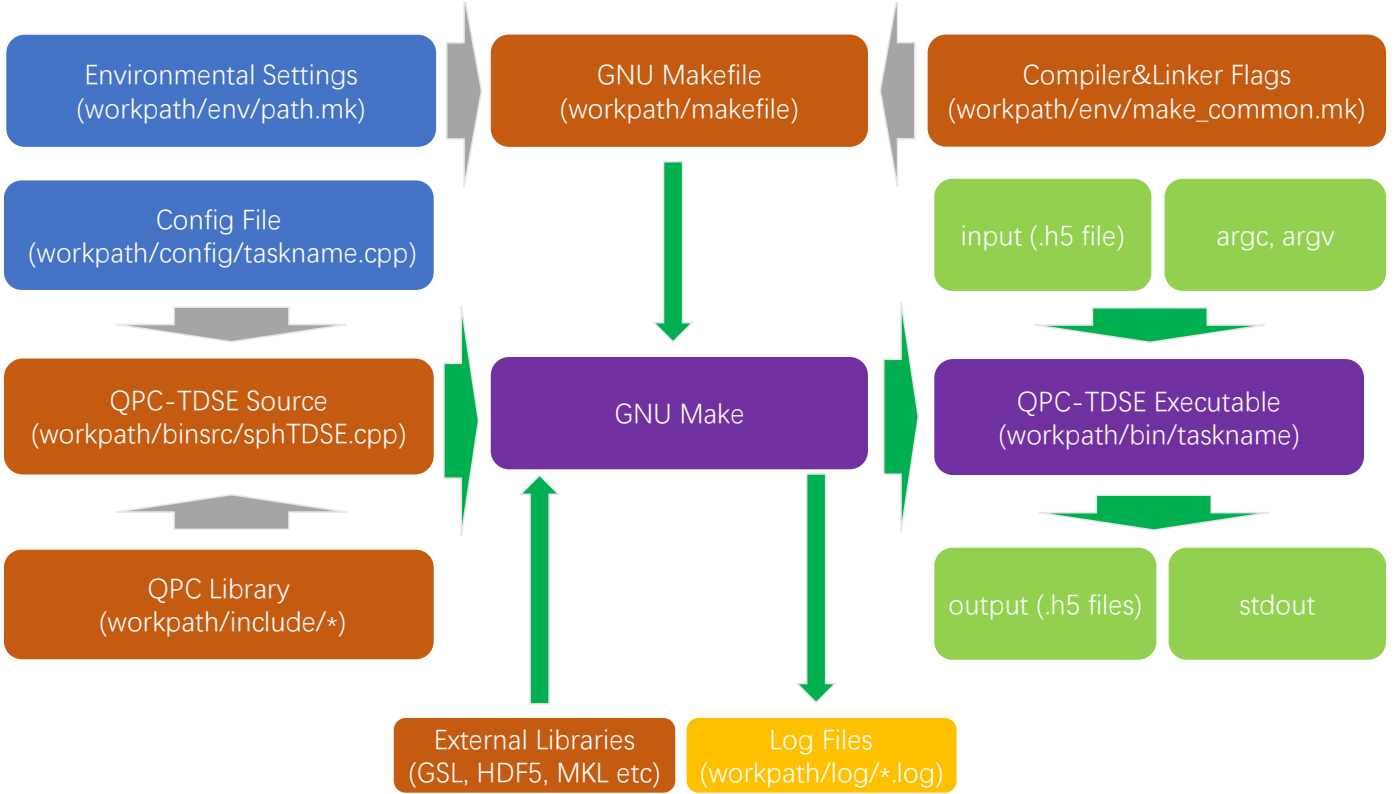


Figure 1: Schematic of How-to-Use. Orange blocks are files that users usually do not need to modify. Blue blocks are input files that should be correctly set before use. Grey arrows means 'include by' and Green arrows indicates the direction of input/output. Grass blocks are input/output into/from the QPC-TDSE executable. Yellow blocks are log files from compiler and linker. Purple blocks are executables that users need to run.

After correctly setting the environmental variables, we may briefly introduce how to use QPC-TDSE and the basic structure of the project. A detailed introduction should be found in the next section. In short, working with QPC-TDSE constitutes four steps:

(a) Modify the compile-time parameters in **workpath/config/taskname.cpp**;

QPC-TDSE treats some of the never-modified parameters (e.g. grid constant, box size) as compile-time constants (constexpr variables in C++) to benefit from compile-time optimization. These variables are defined and given values in configuration files written in C++ grammar and will be included in and compiled with the QPC-TDSE launching code **workpath/binsrc/sphTDSE.cpp**. Here **taskname** is chosen at user's wish, as long as it is a legal name for linux file. In principle, you never need to modify **workpath/binsrc/sphTDSE.cpp** and the header-only library behind it directly for standard use.

- (b) Build the binary executable **workpath/bin/taskname**;

In **workpath**, a makefile is provided to build it. Type

```
make -B taskname
```

is all you need to do. This typically takes one or two seconds. If the compilation is successful, you will see a welcome message and **workpath/bin/taskname** will be generated. Otherwise, the error message of the compiler and linker will be redirected to **workpath/log/cc\_error.log** and **workpath/lod/ld\_error.log**. Failures could occur if users give incorrect values to the parameters; or no **workpath/config/taskname.cpp** is found; or the installation/external libraries/environmental variables of the system are not correctly done. By default the makefile will search **workpath/config/sphTDSE.cpp** and generate **workpath/bin/sphTDSE**. Running

```
make clean
```

would clean up everything in **workpath/bin** and **workpath/log** without check, even though they are not generated by the project, so usual files should never be placed in these directories.

- (c) Run by parsing dynamical parameters to the binary **workpath/bin/taskname**;

QPC-TDSE reads some of the dynamical parameters (e.g. laser parameters, path of to-be-saved output data) from argv. If you run **workpath/bin/taskname** without arguments, a manual about their expected format will be printed to stdout. The output files will be overwritten without asking so be careful with the name of output files.

- (d) Conduct analysis with the output file.

QPC-TDSE itself only produces raw data and leaves the analysis and interpretation of the results to the user.

The above procedures are also illustrated in fig.1. An introduction of the outermost directory structure is given next.

**env** stores the mk files designating enviromental variables and the compiler flags.

**include** stores the QPC header-only library, most of which having .h or .hpp suffix.

**binsrc** stores the source launching codes (.cpp) to be compiled.

**config** stores the compile-time parameter files.

**script** stores the bash scripts to launch example codes.

**bin** stores binary files compiled from sources in **binsrc**.

**log** stores the messages printed by GCC.

## 2.4 Example Runs

We provide some simple example for using QPC-TDSE. They could also be used as testruns to check whether the programme are working as expected. Each example contains two file: a **.cpp** file in **config** given name with the syntax **exampleXXX.cpp**, and a launching bash script file in **script** given name with the syntax **exampleXXX.sh**. The former contains a (very long) list of parameters and settings written in C++ assignment, for example

```
static constexpr double para_rmax = 240.; \\ assign value to para_rmax
```

The exact meaning of these parameters will be given in next section. Being a C++ file implies that when you edit it you must follow C++ grammar (it only use very simple ones, don't panic) and if you do something wrong, the compiler will complain about it instead of QPC-TDSE. Indeed, QPC-TDSE is not responsible for telling what you do wrong in C++ coding. However it does recognize some of obviously unphysical parameters and will forbid you from compiling the code with them through the static assert statements.

Before running an example, it is suggested to check the parameters in the corresponding config file because sometimes the resources it requested could exceed the limit your system can provide. To run example01-a-gs, the only thing you need to do is to cd into your workpath and run the script by sh

```
sh ./script/example01_a_gs.sh
```

or run it after authorization



```
chmod a+x ./script/example01_a_gs.sh && ./script/example01_a_gs.sh
```

To see what exactly these scripts do, you may simply open it by any text editor. The script `./script/example01_a_gs.sh` will first compile `./bin/example01_a_gs` with the config code `./config/example01_a_gs.cpp` and execute it with appropriate arguments.

Note that the example scripts are only for test and illustration, thus not designed in a thread-safe manner. Namely, if you run the same example script from two different processes simultaneously, the behavior is undefined. If you prefer to run multiple tasks concurrently, a feasible way is to compile multiple executables having different names. Nevertheless, running an existing executable is always thread-safe, which means you may call the same executable concurrently for multiple times with different arguments. In this case, do not use the same save path, otherwise outputs could be overwritten, or error could occur when two different processes try to open the same file with non-read-only flags. On the other hand, reading the same input file is safe since QPC-TDSE opens these files with read-only flags.

A package of analysis scripts written in MATLAB can be found in the directory `workpath/analysis`. Among them, the script `load_hdf5.m` loads the data contained in an HDF5 file exported by QPC into your workspace. The script `convert.m` reshapes all the raw data into their original tensor form, after which you may call the rest scripts to plot the results. For example running the scripts `plot_spec_z.m` or `plot_spec_xy.m` will plot the momentum distribution in a predefined manner.

## 2.5 Uninstall

For uninstallation, one may simply remove the whole `workpath`.

## 3 Implementation

### 3.1 Parameter Interpretation

Hereby we list the meaning of the static parameters in a config file.

#### 3.1.1 workflow parameters (1)

Table 1: workflow parameters

Name	Type	Legal Value
to_save_field	int <sup>a</sup>	0,1
calculate_init	int	0,1,2
calculate_prop	int	0,1
calculate_coef	int	0,1
calculate_disp	int	0,1
calculate_proj	int	0,1
calculate_spec	int	0,1,2,3
calculate_tsurff	int	0,1
calculate_obs_v_z	int	0,1
calculate_obs_v_r	int	0,1
use_mask_function	int	0,1

<sup>a</sup> the code has only been tested with `int=32_t`

(1-a) When the flag `to_save_field` is 0, no field value will be saved; Otherwise, field values of  $F_x(t_j)$ ,  $F_y(t_j)$ ,  $F_z(t_j)$  on all time steps (including field-free ones) together with  $t_j$  will be saved as long as they are not switched off [to switch off them see (9)]. Their storage layer could be found in (9-e).

(1-b) When the flag `calculate_init` is 0, QPC-TDSE will prepare the coefficient object  $c_{nlm}$  by reading data from an hdf5 file and ignore any inputs from calling @a or @A [see (10-b)]. In this case, the data presented in the file should satisfy some conditions, see (7-f). A coefficient object saved from a previous run using the same grid parameters [see (2)], of course, satisfy these conditions. Namely you can read data from the file output by another simulation performed on the same grid.

When the flag `calculate_init` is 1, QPC-TDSE will try exact diagonalization approaches to obtain eigenstates of the system to set the coefficient object  $c_{nlm}$ . In this case you must set `init_ns>0` [see (10-a)]. If `n_pole=0` [see (2-d)], which means you are working with a centrifugal potential, the diagonalization is restricted in an r-subspace and done by LAPACKE. Otherwise, it will diagonalize it in the full space by FEAST. To choose the eigenstate(s) you want in either case, you should call @a or @A with proper arguments [see (10-b)].

When the flag `calculate_init` is 2, QPC-TDSE will set the time step used in propagation by  $-i\Delta t$  [see (7)] to perform imaginary time propagation (ITP) initialized with an internally generated guess wavefunction. This works only for `n_pole>0` [see (2-d)], `n_field_x=n_field_y=n_field_z=0` [see (9)], `calculate_prop=1` [see (1-3)] and `para.td>0` [see (7)]. Otherwise the compilation will fail through static asserts. Namely, for atomic system you cannot use ITP currently. And when using it, you should turn off all lasers and enable propagation, and set the free-propagation time to non-negative values.

(1-c) When the flag `calculate_prop` is 0, QPC-TDSE will skip the propagation stage (both with-field and field-free ones). This is typically useful for separation of propagation and analysis. Namely one may firstly run propagation-only and save the coefficient objects, and then run an analysis by loading the result [see (1-b)] and skipping the propagation.

When the flag `calculate_prop` is 1, QPC-TDSE will do propagation after preparing all necessary data. Details see (7).

(1-d) When the flag `calculate_coef` is 1, QPC-TDSE will save the coefficient objects at the end of the programme, regardless whether it is propagated or not. Otherwise, no coefficients will be saved. You may probably disable this when you don't need them. The data layer of coefficient objects to be saved should be found in (2).

(1-e) When the flag `calculate_disp` is 1, QPC-TDSE will compute the values of the wavefunction on requested spatial points [see (4)] according to the coefficient object  $c_{nlm}$  at the end of the programme, regardless whether it is propagated or not.

Otherwise, no wavefunction data will be saved and displayer parameters [see (4)] will be ignored. The data layer of saved wavefunction values is found in (4).

- (1-f) When the flag `calculate_proj` is 1, QPC-TDSE will compute the projection onto eigenstates at the end of the programme, regardless whether it is propagated or not [details found in (6)]. If it is 2, the projection is done during propagation. The flag only works for `n_pole=0` [see (2-d)], since usually only the solution of the full eigensets in centrifugal systems is affordable.

When the flag `calculate_proj` is 0, or `n_pole>0` [see (2-d)], the programme simply ignores this flag and associated parameters [see (6)] during the run.

- (1-g) When the flag `calculate_spec` is 1, QPC-TDSE will compute the angular-resolved photo-electron momentum spectra on requested momentum grid [see (5)] via projection the wavefunction onto the exact scattering states if `n_pole=0` [see (2-d)] or onto the planewave after the application of a heavieside-like operator if `n_pole>0` [see (2-d)], regardless whether propagated or not. One should carefully ensure the necessary wavepackets are still inside the box when using this functionality.

When the flag `calculate_spec` is 2, QPC-TDSE will compute the partial-wave photo-electron momentum spectra on the same requested grid. One may also set the flag to 3 to enable both of them.

When the flag `calculate_spec` is 0, no spectra will be saved and associated parameters [see (5)] will be ignored.

- (1-h) When the flag `calculate_tsurff` is 1, QPC-TDSE will perform t-SURFF calculation during propagation [if enabled, see (1-c)], using the t-SURFF parameters [see (5)]. One typically also needs to set `para_td>0` to get converged results for relatively low-energy part. Be extremely careful that t-SURFF results in QPC-TDSE can not be retrived unless you do the same propagation again. Currently the t-SURFF functionality in QPC-TDSE is only invocable when the laser polarizes either along  $z$ -direction or in  $x - y$  plane.

When the flag `calculate_tsurff` is 0, t-SURFF will be turned off and all associated parameters [see (5)] will be ignored. The data layer of saved results are the same to the spectrum in (1-g), and should be found in next section.

- (1-i) When the flag `calculate_obsz` is 1, QPC-TDSE will compute the  $z$ -observable values  $z(t_s)$  on each time steps as specified in (8) during propagation [if enabled, see (1-c)], together with the time step  $t_s$ . Be extremely careful that observables computed in QPC-TDSE can not be retrived unless you do the same propagation again.

When the flag `calculate_obsz` is 0, no  $z$ -observable will be computed and associated parameters [see (8)] will be ignored. The data layer of saved results is found in next section.

When the flag `calculate_obsr` is 1, QPC-TDSE will compute the  $xy$ -observable values  $x(t_s), y(t_s)$  on each time steps as specified in (8) during propagation [if enabled, see (1-c)], together with the time step  $t_s$ . Be extremely careful that observables computed in QPC-TDSE can not be retrived unless you do the same propagation again.

When the flag `calculate_obsr` is 0, no  $xy$ -observable will be computed and associated parameters [see (8)] will be ignored. The data layer of saved results is found in next section.

- (1-j) When the flag `use_mask_function` is 1, QPC-TDSE will enable the absorber during propagation [if enabled, see (1-c)] according to the parameters specified in (7). Otherwise no absorber will be used, and associated parameters [see (7)] will be ignored.

### 3.1.2 grid and basis parameters (2)

- (2-a) `para_rmin` and `para_rmax` specifies the range of simulation box  $r_{\min}$  and  $r_{\max}$ , where for the current spherical box `para_rmin` must be set to 0. The value of `para_rmax`, however, could vary from case to case and its reasonableness must be carefully identified by the user.
- (2-b) `n_rank` specifies the rank of the B-spline basis functions used in the simulation, i.e.  $M$ . We strongly recommend to use `n_rank=8` because the current code does some special optimization for that case, although using other values is still acceptable.
- (2-c) `n_ndim` specifies the number of active radial B-spline basis in the simulation, i.e.  $N_r$ . Here 'active' means 'not excluded from the full B-spline set', so the two removed B-spline functions which takes non-zero value on boundary  $r_{\min}$  and  $r_{\max}$  [see (2-a)] are referred to as 'inactive' and is not count in. To avoid conceptual errors, `n_ndim` must be larger than  $2 \times n\_rank$  for the current code (usually you don't need to worry for that because practical simulations use far more basis).

Table 2: grid and basis parameters

Name	Type	Legal Value
para_rmin	double	0.
para_rmax	double	> 0
n_rank	size_t	see text
n_ndim	size_t <sup>a</sup>	see text
n_pole	size_t	see text
n_knot	std::array<size_t,n_pole>	see text
n_ldim	size_t	> 1
n_mdim	size_t	> 0
n_mmin	long <sup>b</sup>	all
n_core	size_t	> 0

<sup>a</sup> the code has only been tested with size\_t=uint64\_t<sup>b</sup> the code has only been tested with long=int64\_t

The current code assign values to the knots using a linearly increasing sequence between radial poles [see (2-d)]. The knot sequence is saved into the file at the end of each run, given the name **/knot**. The value of n\_ndim is associated with the convergency of the simulation, and is explained at the end of section 3.2 in the main text of the paper.

It should be mentioned that it is preferable to set  $N_r$  to be a multiple of 4 such that any pointer being shifted from the head (which is aligned on a 512 byte memory boundary) by a multiple of  $N_r$  keep their alignment and possible optimization could happen for the external libraries, e.g. the MKL BLAS. The QPC itself does not explicitly align the stride of the innermost dimension when  $N_r$  is not a multiple of 4 and just leaves everything in their natural way. For similar reasons, there could be 'magic numbers' of  $N_r$  using which the programme runs much faster than nearby values, e.g.  $N_r = 512$  could be faster than  $N_r = 507$ .

- (2-d) n\_pole specifies the number of radial poles (c.f. physical poles)  $N_{\text{pole}}$ . For a pure centrifugal potential scenario, it should be set to 0. A 'physical pole' is an individual Coulomb center which exists physically off from the  $r = 0$  origin. Multiple physical poles having the same distance to the origin will share the same radial pole for the consideration that less matrix elements are required. To specify the parameters of all physical poles, call the initializer @P in the proper format:

```
<executable> <otherflags> @P0 <R0> <Z00> <TH00> <PH00> \
                                <Z01> <TH01> <PH01> \
...
                                @P1 <R1> <Z10> <TH10> <PH10> \
...
```

Here R-s should be positive real numbers (in atomic unit) to assign values to the radial pole  $\{R_\nu\}_{\nu=0}^{N_{\text{pole}}-1}$ . Z-s, TH-s, PH-s should be real numbers to assign values to  $\{Z_{\nu,s}, \theta_{\nu,s}, \varphi_{\nu,s}\}_{\nu=0}^{N_{\text{pole}}-1}$ , which describe the property of physical Coulomb poles. Z-s are charge numbers, TH-s and PH-s are polar angles of the physical poles, given in unit of  $\pi$ . At least one physical pole should be specified for each radial pole. If you set n\_pole>0 without calling @P, the behavior of the programme is undefined. The radial axis are thus isolated by radial poles to form  $1+n_{\text{pole}}$  intervals, namely

$$(r_{\min}, R_0), (R_0, R_1), \dots, (R_{N_{\text{pole}}-1}, r_{\max}).$$

n\_knot is an integer array which specifies the number of knot points inside each interval, i.e.  $\{K_\nu\}_{\nu=0}^{N_{\text{pole}}-1}$ . Given n\_knot, a set of real numbers  $\{r_s\}_{s=0}^N$  is automatically computed to formulate the B-spline knot sequence. As an example, if one set n\_pole=1, the knots will be like

$$\underbrace{r_0, r_0, \dots, r_0}_{M}, \underbrace{r_0, r_1, \dots, r_{K_0-1}}_{K_0}, \underbrace{r_{K_0}, r_{K_0}, \dots, r_{K_0}}_{M-1}, \underbrace{r_{K_0}, r_{K_0+1}, \dots, r_{N-1}, r_N}_{\text{automatically computed}}, \underbrace{r_N, r_N, \dots, r_N}_M \quad (1)$$

Legal values for  $K_\nu$  should satisfy  $M + \sum_\nu K_\nu + N_{\text{pole}}(M-1) < N_r + 2$ .

- (2-e) n\_ldim, n\_mdim and n\_mmin specifies the dimension of the spherical harmonic grid in the simulation, i.e.  $N_l$ ,  $N_m$  and  $m_0$ , respectively. Their necessary values strongly depend on the physical scenario and could vary from a few to a few hundreds.

Similar to  $N_r$ , it is favorable for the underlying linear algebra libraries to gain possible boost by setting the product  $N_l N_m$  to be a multiple of 4, e.g. using  $N_l = 40$  could be much faster than using  $N_l = 39$ .

- (2-f) `n_core` specifies the OpenMP threads to be used throughout the lifetime of the programme. Do not confuse it with the number of physical core or logical core on your system. Basically we does not suggest to use `n_core` larger than the number of logical cores on your sytem. In principle, setting `n_core` to the number of physical cores will mininize the wall-clock time for the simulations on very large grids. This is typically not the case for those with small and even moderate grids, where too many threads could waste your computer resources on the overheads of communication. To maximize the performance of the propragmme, we suggest that parameters should satisfy  $\text{mod}(\text{n\_mdim} \times \text{n\_ldim}, 4 \times \text{n\_core}) = 0$  such that the workload of each thread is balanced for the Crank-Nicolson kernels. The code calls the BLAS library in MKL, which is internally parallelized with `n_core` threads, thus 'magic numbers' of `n_core` could also exist.

Besides, please notice that the memory consumption of some functionalities in QPC-TDSE could grow linearly with respect to `n_core`.

The complex-valued coefficient object  $c_{nlm}$  are stored in an one-dimensional array, with its index vector  $(i_m, i_l, i_r)$  mapped to the one-dimensional index:

$\text{Re}[c(i_m, i_l, i_r)]$  is stored in  $2(i_m N_r N_l + i_l N_r + i_r)$ ;

$\text{Im}[c(i_m, i_l, i_r)]$  is stored in  $2(i_m N_r N_l + i_l N_r + i_r) + 1$ ;

The length of the array is  $2N_r N_l N_m$ . In the hdf5 file, it will be named `/coef1`. All associated grid parameters will also be saved.

### 3.1.3 potential parameters (3)

Table 3: potential parameters

Name	Type	Legal Value
<code>potn</code>	callable objects double(double)	see text
<code>para_zasy</code>	double	see text
<code>para_rasy</code>	double	see text

- (3-a) `potn` specifies the centrifugal potential  $V(r)$ . It should be a callable object convertible into `std::function<double(double)>`, for example a lambda expression:

```
static constexpr auto potn = [](double r){return -1./(r+1e-50);};
```

Some pre-defined potential models are given in `include/spherical/potential.hpp`. For example you may use the pre-defined hydrogenic potential with  $Z = 1.0$  by

```
static constexpr auto potn = spherical::hydrogenic{1.0};
```

In principle, any well-behaved potentials may be accepted as legal values. 'Well-behaved' means the return value by calling the function with  $r_{\min} \leq r \leq r_{\max}$  should not be nan, inf or extreme values near overflow or underflow limit of double-precision floating point numbers, and the potential it describes should be a mathematically smooth function.

- (3-b) `para_zasy` and `para_rasy` are hint values for the radial ODE solver. They are only used for non-predefined potentials when the flag `calculate_spec`  $\neq 0$  and ignored when a pre-defined potential is used. They specifies the asymptotical behavior of the potential, i.e.

$$\lim_{r \rightarrow \infty} rV(r) = Z_a,$$

and  $r_a$  is your suggested value beyond which the solution to the radial ODE could be practically recognized as inside asymptotical region. The solver will choose the maximum value of  $r_a$  and  $(\sqrt{l(l+1)} + (Z_a/k)^2 - Z_a/k)/k$ . If your  $V(r)$  does not satisfy the asymptotic behavior required by QPC-TDSE but you still enables the functionalities of PMD, the programme's behavior will be undefined.

Table 4: wavefunction displayer parameters

Name	Type	Legal Value
disp_rmin	double	$\geq 10^{-8}$
disp_rmax	double	$\leq r_{\max}$
disp_thmin	double	$\geq 0$
disp_thmax	double	$\leq \pi$
disp_phmin	double	$\geq 0$
disp_phmax	double	$\leq 2\pi$
disp_nr	size_t	$\geq 1$
disp_nth	size_t	$\geq 1$
disp_nph	size_t	$\geq 1$

### 3.1.4 wavefunction displayer parameters (4)

The displayer parameters describe a spherical mesh on which the wavefunction values are to be evaluated if calculate\_disp=1. The mesh points are:

$$r_i = \text{disp\_rmin} + i \times (\text{disp\_rmax} - \text{disp\_rmin}) / (\text{disp\_nr} - 1) \text{ if } \text{disp\_nr} > 1$$

$$r_i = \text{disp\_rmin} \text{ if } \text{disp\_nr} = 1$$

$$\theta_i = \text{disp\_thmin} + i \times (\text{disp\_thmax} - \text{disp\_thmin}) / (\text{disp\_nth} - 1) \text{ if } \text{disp\_nth} > 1$$

$$\theta_i = \text{disp\_thmin} \text{ if } \text{disp\_nth} = 1$$

$$\varphi_i = \text{disp\_phmin} + i \times (\text{disp\_phmax} - \text{disp\_phmin}) / (\text{disp\_nph} - 1) \text{ if } \text{disp\_nph} > 1$$

$$\varphi_i = \text{disp\_phmin} \text{ if } \text{disp\_nph} = 1$$

The complex wavefunction values  $\psi(r_i, \theta_j, \varphi_k)$  will be stored in a one-dimensional double array according to the index map:

The index of  $Re[\psi(r_i, \theta_j, \varphi_k)]$  is  $2 \times (k \times \text{disp\_nr} \times \text{disp\_nth} + j \times \text{disp\_nr} + i)$ ,

The index of  $Im[\psi(r_i, \theta_j, \varphi_k)]$  is  $2 \times (k \times \text{disp\_nr} \times \text{disp\_nth} + j \times \text{disp\_nr} + i) + 1$ .

The length of this array is  $2 \times \text{disp\_nr} \times \text{disp\_nth} \times \text{disp\_nph}$ . In the hdf5 file, it will be named **/wave1**. All associated grid parameters will also be saved.

### 3.1.5 momentum distribution parameters (5)

Table 5: momentum distribution parameters

Name	Type	Legal Value
spec_krmin	double	see text
spec_krmax	double	see text
spec_thmin	double	$\geq 0$
spec_thmax	double	$\leq \pi$
spec_phmin	double	$\geq 0$
spec_phmax	double	$\leq 2\pi$
spec_nkr	size_t	$\geq 1$
spec_nkt	size_t	$\geq 1$
spec_nkp	size_t	$\geq 1$
spec_axis_type	int	0,1
tsurff_zc	double	0.
tsurff_r0	double	see text

(5-a) `tsurff_zc` is a value reserved for future use. Currently it should be set to 0.

If `n_pole=0` and t-SURFF is enabled, `tsurff_r0` specifies the radius of t-SURFF boundary. It should be smaller than  $r_{\max}$  and sufficient far from the absorption region. For physical considerations, it should also be much larger than the electron's quiver radius and sufficient far from the Coulomb center(s). Its value should be specified empirically by the user.

If `n_pole>0` and projection method is enabled, `tsurff_r0` specifies the position of the step function which effectively projects out all bound states.

In other cases, both parameters will be ignored.

#### (5-b) angular-resolved PMD storage

Computed via either projection approach or t-SURFF, the output  $P(\tilde{k}, \tilde{\theta}, \tilde{\varphi})$  will share the same data layer. The mesh points are:

$$\begin{aligned}\tilde{k}_i &= \text{spec\_krmin} + i \times (\text{spec\_krmax} - \text{spec\_krmin}) / (\text{spec\_nkr} - 1) \text{ if } \text{spec\_nkr} > 1 \text{ and } \text{spec\_axis\_type}=0 \\ \frac{\tilde{k}_i^2}{2} &= \frac{\text{spec\_krmin}^2}{2} + i \times \left( \frac{\text{spec\_krmax}^2 - \text{spec\_krmin}^2}{2} \right) / (\text{spec\_nkr} - 1) \text{ if } \text{spec\_nkr} > 1 \text{ and } \text{spec\_axis\_type}=1 \\ \tilde{k}_i &= \text{spec\_krmin} \text{ if } \text{spec\_nkr} = 1 \text{ and } \text{spec\_axis\_type}=0 \\ \frac{\tilde{k}_i^2}{2} &= \frac{\text{spec\_krmin}^2}{2} \text{ if } \text{spec\_nkr} = 1 \text{ and } \text{spec\_axis\_type}=1 \\ \tilde{\theta}_i &= \text{spec\_thmin} + i \times (\text{spec\_thmax} - \text{spec\_thmin}) / (\text{spec\_nth} - 1) \text{ if } \text{spec\_nth} > 1 \\ \tilde{\theta}_i &= \text{spec\_thmin} \text{ if } \text{spec\_nth} = 1 \\ \tilde{\varphi}_i &= \text{spec\_phmin} + i \times (\text{spec\_phmax} - \text{spec\_phmin}) / (\text{spec\_nph} - 1) \text{ if } \text{spec\_nph} > 1 \\ \tilde{\varphi}_i &= \text{spec\_phmin} \text{ if } \text{spec\_nph} = 1\end{aligned}$$

The complex-valued  $P(\tilde{k}_i, \tilde{\theta}_j, \tilde{\varphi}_k)$  will be stored in a one-dimensional double array according to the index map:

The index of  $Re[P(\tilde{k}_i, \tilde{\theta}_j, \tilde{\varphi}_k)]$  is  $2 \times (i \times \text{spec\_nth} \times \text{spec\_nph} + j \times \text{spec\_nph} + k)$ ,

The index of  $Im[P(\tilde{k}_i, \tilde{\theta}_j, \tilde{\varphi}_k)]$  is  $2 \times (i \times \text{spec\_nth} \times \text{spec\_nph} + j \times \text{spec\_nph} + k) + 1$ .

The length of the array is  $2 \times \text{spec\_nkr} \times \text{spec\_nth} \times \text{spec\_nph}$ . The saved arrays are named with **pmd1** for projection approach and **pmd2** for t-SURFF approach. All associated grid parameters will also be saved.

#### (5-c) partial-wave PMD storage

Currently in QPC-TDSE the partial-wave PMD can only be obtained with projection approach. The output  $P(\tilde{k}, m, l)$  has the same mesh points of  $\tilde{k}$  to (5-b). The index of  $m, l$  follows a similar way of coefficient grid, namely by  $m = i_m + m_0$ ,  $l = i_l + |m|$ . The complex-valued  $P(\tilde{k}_i, i_m, i_l)$  will be stored in a one-dimensional double array according to the index map:

The index of  $Re[P(\tilde{k}_i, i_m, i_l)]$  is  $2 \times (i \times N_m \times N_l + i_m \times N_l + i_l)$ ,

The index of  $Im[P(\tilde{k}_i, i_m, i_l)]$  is  $2 \times (i \times N_m \times N_l + i_m \times N_l + i_l) + 1$ .

The length of the array is  $2 \times \text{spec\_nkr} \times N_m \times N_l$ , with name **lmd1**. All associated grid parameters will also be saved.

For photo-energy spectra  $P(E)$ , you may sum up all partial waves. This would be more accurate than numerical integrating over the angular-resolved PMD.

For the consideration of numerical stability, `spec_krmin` should not be too small in the projection approach. `spec_krmin > 0.01` has been tested to show satisfactory accuracy. The number of sample points `spec_nkr` should not be too large, otherwise the unphysical oscillation could occur in the obtained spectrum, which is induced by using pseudo-continuum in a box instead of genuine ones. Besides, for t-SURFF to converge, a sufficient large `para_td` [see (7)] should be used for small `spec_krmin`, which can be estimated via `tsurff_r0/spec_krmin`.

Table 6: eigenstate projection parameters

Name	Type	Legal Value
proj_samp_rate	size_t	$\geq 1$
proj_filter	callable object	see text

### 3.1.6 eigenstate projection parameters (6)

Only @L and @E [see (7-f)] are equipped with this functionality. If calculate\_proj is set nonzero, the projection onto field-free eigenstates will be computed. If it is 2, it is done per proj\_samp\_rate step(s) during propagation (including free propagation). If it is 1, it is only done at the end of programme. To choose the eigenstate you are interested in, you shall specify the callable object proj\_filter. The following example chooses all states with  $l \leq 2$  and  $E < -0.05$ :

```
static constexpr auto proj_filter = [](size_t l, double e, double* v)
{
    return e < -0.05 && l <= 2;
};
```

More involved forms of filters will be released in future version. The output file contains two or three objects **proj\_val**, **proj\_nnz** and optionally **proj\_samp\_rate**. **proj\_nnz** is a one dimensional array with size  $N_m N_l$ . The  $(i_m N_l + i_l)$ -th element of **proj\_nnz** stores the number of selected states in subspace  $lm$ . The projection values are stored in another one dimensional double array **proj\_val**. The real and imaginary part of projection onto  $i_n$ -th state inside the  $I$ -th  $lm$ -subspace, where  $I = i_m N_l + i_l$ , is stored in  $2J$  and  $2J + 1$ , where  $J = i_n + \sum_{i=0}^{I-1} \mathbf{proj\_nnz}(i)$ .

### 3.1.7 propagation parameters (7)

Table 7: propagation parameters

Name	Type	Legal Value
para_dt	double	$> 0$
para_td	double	$\geq 0$
propflag_eve	bool	see text
propflag_odd	bool	see text
mask_rmin	double	$\geq \text{para\_rmin}$
mask_rmax	double	$\geq \text{para\_rmax}$
mask_fact	double	$> 0$
mask_nthd	size_t	see text

- (7-a) para\_dt specifies the time step  $\Delta t$  for the Crank-Nicolson propagator. Typical values could vary from 0.005 to 0.1. If  $\Delta t$  is too large, unconverged or even unphysical behavior could occur in the results. Such situation can not be automatically detected by codes and should carefully checked by the user.
- (7-b) para\_td specifies the duration of the free-propagation stage. Extra  $\text{floor}(\text{para\_td}/\text{para\_dt})$  steps will be computed after the with-field propagation.
- (7-c) mask\_rmin and mask\_rmax specifies the range of the absorption region. Economic choice should let mask\_rmax and para\_rmax equal.
- (7-d) mask\_fact specifies the index  $\alpha$  in the mask function  $\cos^\alpha$ . To have invariant simulation results for different  $\Delta t$ ,  $\alpha$  should be propotional to  $\Delta t$ .
- (7-e) mask\_nthd specified the number of threads to run the absorber.
- (7-f) The propagation using different propagators shall be specified at runtime. To call the one specialized for LP lasers in z-direction and pure centrifugal potential, use @L:

```
<executable> <otherflags> @L <load_path> <save_path>
```



To call the one specialized for EP lasers in xy-plane and pure centrifugal potential, use @E:

`<executable> <otherflags> @E <load_path> <save_path>`

To call the one specialized for LP or EP lasers and multipolar potential, use @O:

`<executable> <otherflags> @O <load_path> <save_path>`

load\_path is the path of file, which should contain a coefficient object. In case calculate\_init≠0 such that you do not need to load, you may set load\_path to arbitrary values. If it is zero, the grid parameters [see (2)] in the config file from which the binary is compiled from and the grid parameters presented in the to-be-loaded file should follow the constraints:

1. n\_ndim, n\_rank, para\_rmin, para\_rmax must be equal.
2. to-be-loaded coefficient object should be on a smaller lm grid which can be fit onto the new grid.

The save\_path is the path of file to save all the results of the run. Be careful that the programme overwrites files with the same name without asking.

(7-g) propflag\_even and propflag\_odd are switches to turn on/off the propagation of odd and even space for @E. They can not be set false simultaneously, and will be ignored by @L and @O.

### 3.1.8 observable parameters (8)

Table 8: observable parameters

Name	Type	Legal Value
obsv_func_z	callable object	see text
obsv_func_r	callable object	see text

Dipolar observables are defined via

$$z(t) = \int \psi^*(\mathbf{r}t) f(r) \cos(\theta) \psi(\mathbf{r}t) d\mathbf{r} \quad (2)$$

$$r_{\pm}(t) = \int \psi^*(\mathbf{r}t) f(r) \sin(\theta) \exp(\pm i\varphi) \psi(\mathbf{r}t) d\mathbf{r} \quad (3)$$

and  $x(t) = [r_+(t) + r_-(t)]/2$ ,  $y(t) = [r_+(t) - r_-(t)]/2i$ . In above definition, the only thing you need to take care is the function  $f(r)$ . It should be a callable object convertible into `std::function<double(double)>`, for example a lambda expression:

```
static constexpr auto obsv_func_z = [](double r){return r;};
```

obsv\_func\_z will be used to evaluate  $z(t)$ , and obsv\_func\_r will be used to evaluate  $x(t)$ ,  $y(t)$ . For dipole acceleration in centrifugal potential model, one may set them by

$$\nabla V(r) = \frac{dV(r)}{dr} \frac{\mathbf{r}}{r}. \quad (4)$$

The values of observables at  $\{t_s\}_{s=0}^{s=N_t-1}$  will be saved in a one dimensional double array whose size is  $N_t$  [see (9)] if the corresponding flags are turned on [see (1-i)]. Together saved are the values of  $\{t_s\}_{s=0}^{s=N_t-1}$ . Their names are `/x_list`, `/y_list`, `/z_list` and `/t_list`.

### 3.1.9 laser field parameters (9)

QPC-TDSE currently adopts the laser model

$$F_s(t) = \sum_{i=0}^{N_s-1} F_{si}(t), \quad (5)$$

with  $s = x, y, z$ .

(9-a) n\_field\_x, n\_field\_y, n\_field\_z specifies the number of lasers components  $N_x$ ,  $N_y$ ,  $N_z$ . When they are set to zero, the laser in corresponding direction is turned off. The value of  $F_x$  and  $F_y$  will be discarded in @L, and the value of  $F_z$  will be discarded in @E.

Table 9: laser field parameters

Name	Type	Legal Value
n_field_x	size_t	all
n_field_y	size_t	all
n_field_z	size_t	all
gauge	int	0,1
envelope_type	type alias	see text

(9-b) The laser model is chosen by `envelope_type`. It should be a type alias to the class name which implements  $F_{si}(t)$ . We provide several pre-defined models in **include/structure/field.h**. To work with them, use

```
using envelope_type = qpc::field_component_sinen<N>; //sine power N
```

such that

$$F_{si}(t) = F_0 \Theta(\omega t - 2\pi N_{\text{start}}) \Theta(2\pi(N_{\text{start}} + N_{\text{dura}}) - \omega t) \sin^N\left(\frac{\omega t - 2\pi N_{\text{start}}}{2N_{\text{dura}}}\right) \sin(\omega t + \phi_{\text{CEP}}), \quad (6)$$

or

```
using envelope_type = qpc::field_component_gauss;
```

such that

$$F_{si}(t) = F_0 \exp(-2 \ln 2 \left(\frac{\omega t/2\pi - N_{\text{center}}}{N_{\text{FWHM}}}\right)^2) \sin(\omega t + \phi_{\text{CEP}}), \quad (7)$$

or

```
using envelope_type = qpc::field_component_trapz<N1,N2>; //N1 rise, N2 fall
```

such that

$$F_{si}(t) = F_0 T(N_1, N_{\text{dura}} - N_2, N_{\text{dura}}; \frac{\omega t}{2\pi} - N_{\text{start}}) \sin(\omega t + \phi_{\text{CEP}}), \quad (8)$$

where

$$T(N_a, N_b, N_c; x) = \begin{cases} \frac{x}{N_a}, & N_a > x > 0 \\ 1, & N_b \geq x \geq N_a \\ \frac{N_c - x}{N_c - N_b}, & N_c > x > N_b \\ 0, & \text{otherwise} \end{cases} \quad (9)$$

If a user-customized field model is in need, you may write some C++ in the config file to define it. Any user defined laser function should implements the following interfaces:

```
struct your_laser_field
{
    double tmin();
    double tmax();
    double operator()(double);
    void initialize(double, double, double, double, double);
};
using envelope_type = your_laser_field;
```

Here `tmin` and `tmax` should be implemented as the minimal and maximal value of  $t$  at which the field amplitude can not be considered as vanishing. The programme collects and compares all values returned by `tmin` and `tmax` to get a global  $t_{\text{min}}$  and  $t_{\text{max}}$ , and determine the number of steps for with-field propagation  $N_t = \text{floor}((t_{\text{max}} - t_{\text{min}})/\Delta t)$ . Calling the `operator()` with time  $t$  should return the field value at this instance. `initialize(double, double, double, double, double)` should be implemented as its initializer which takes value from `arg0-arg4`. Also, the object of this class should be trivially constructible.

(9-c) Laser parameters are also parsed in by command line arguments. For example, to set a single color field  $F_{z0}$ ,

```
<executable> <otherflags> @F Z00 <arg0> Z01 <arg1> Z02 <arg2> \
Z03 <arg3> Z04 <arg4>
```

The first integer in  $Z_{ij}$  is the index  $i$  of  $F_{si}$ . The second integer  $j$  is the internal index for parameters. Current design allows five (double-precision) parameters at most for each  $F_{si}$ . The meaning of each parameter for the pre-defined laser models are given in tab.10. If the programme runs with uninitialized laser parameters, its behavior is undefined.

Table 10: interpretation of laser parameters and their units

laser model	arg0 (a.u.)	arg1 (a.u.)	arg2 (o.c.)	arg3 (o.c.)	arg4 ( $\pi$ )
field_component_sinen< $N$ >	$F_0$	$\omega$	$N_{\text{dura}}$	$N_{\text{start}}$	$\phi_{\text{CEP}}$
field_component_gauss	$F_0$	$\omega$	$N_{\text{FWHM}}$	$N_{\text{center}}$	$\phi_{\text{CEP}}$
field_component_trapz< $N_1, N_2$ >	$F_0$	$\omega$	$N_{\text{dura}}$	$N_{\text{start}}$	$\phi_{\text{CEP}}$

(9-d) Setting gauge=0 will choose velocity gauge throughout the programme.  $F_s$  will be interpreted as vector potential  $A_s$ , and the velocity gauge propagator will be called in @L, @E and @O. On the other hand, setting gauge=1 will choose length gauge,  $F_s$  will be interpreted as electric field  $E_s$  and length gauge propagators will be called. It will extraly forbids the user from using t-SURFF through static asserts.

(9-e) If to\_save\_field [see (1-a)] is turned on, all field values will be saved in one-dimensional arrays, with the name /fx\_list, /fy\_list, /fz\_list. All arrays has the same size  $N_t$ .

### 3.1.10 initial state parameters (10)

When the diagonalization approaches are enabled by setting calculate\_init=1 [see (1-3)], QPC-TDSE will initialize the coefficient object  $c_{nlm}$  which fully describes the initial state  $\psi_{\text{init}}$  by

$$\psi_{\text{init}} = \sum_{i=0}^{N_{\text{state}}-1} C_i \psi_i. \quad (10)$$

If you leave any of the  $\psi_i$  unchosen, the behavior of the programme is undefined. For calculate\_init $\neq$ 1 all parameters and functionalities in this item will be ignored.

(10-a) init\_ns specifies the number of states  $N_{\text{state}}$ .

(10-b) To order the state  $\psi_i$  and set values to  $C_i$ , you have to parse arguments to the executable by either

```
<executable> <otherflags> @a <i> <im> <il> <in> <ReC> <ImC>
```

or

```
<executable> <otherflags> @A <i> <El> <Eh> <id> <ReC> <ImC>
```

Here  $i$  should be an integer within the range  $0 \leq i \leq N_{\text{state}} - 1$ . For the initializer @a, the integer  $i_m, i_n, i_n$  is interpreted into quantum numbers according to

$$m = m_0 + i_m, l = |m| + i_l, n = l + 1 + i_n. \quad (11)$$

Here  $m_0$  is specified in (2-e),  $n$  coincides with the main quantum number for hydrogen-like atoms, while for non-hydrogenic ones, it is just an index to numerate the states inside each subspace, rather than the quantum number with defects (which is usually not an integer). Escape character 'z' for  $i_m$  stands for setting  $m = 0$  if  $m_0 \leq 0$  and  $m = m_0$  if  $m_0 > 0$ . Legal values for  $i_m, i_l, i_n$  are  $0 \leq i_m \leq N_m - 1$ ,  $0 \leq i_l \leq N_l - 1$  and  $0 \leq i_n \leq N_r - 1$ . In case illegal values are parsed in, the programme terminates with an error.

For the initializer @A, the double precision floating number  $E_l$  and  $E_h$  selects all states within energy interval  $E_l < E < E_h$ , and integer  $i_d \geq 0$  chooses the  $i_d$ -th state from them using zero-based indexing. When no states are found between  $E_l < E < E_h$ , or  $i_d$  exceeds the number of states found in that interval, or illegal values of  $E_l, E_h$  or  $i_d$  are parsed in, the programme terminates with an error.

ReC and ImC specifies the real and imaginary part of coefficients  $C_i$ . Note that the progragmme will not do renormalization and your initial state will prepared as-is. Default values for ImC is 0.0, and escape character 'o' for ReC stands for 1.0.

## 3.2 Library Structure

In this section we explain the structure of the header-only part of QPC-TDSE, which can be found in the directory **include** in your workpath.

### 3.2.1 include/functions

The code in this directory implements the evaluation algorithm for some special functions, for example the spherical harmonics, the coulomb wave function, the B-spline functions etc, by either calling an external library (e.g. GSL) or by hand-written codes translated from published articles (see the .h files to find the references).

### 3.2.2 include/utilities

Some common functionalities used throughout the code, including exception handler, definition of constants, high dimensional array in C++ style, meta programming tricks etc.

### 3.2.3 include/libraries

Some wrappers to call the external libraries, e.g. HDF, GSL. Some wrappers with SIMD intrinsics are also placed here, although for historical reasons some of them are never going to be used.

### 3.2.4 include/resources

A naïve polymorphic memory allocator which allows memory alignment and native exception handle.

### 3.2.5 include/algorithm

Isolated numerical recipes, like Runge-Kutta solver. Diagonalization procedures are now wrapped to oneAPI calls.

### 3.2.6 include/structure

This directory is named with 'structure' only for historical reasons. There are two files in it. **field.h** implements the pre-defined laser models, while **csrmat.h** wraps function calls to the DSS and FEAST using CSR3 format.

### 3.2.7 include/spherical

The major part of spherical TDSE solver. The only thing you need to call QPC-TDSE in your own source code actually only needs to include **include/spherical/inventory.h** in your code. All QPC header codes are in namespace qpc and all spherical TDSE solver codes are in namespace qpc::spherical. Inside it, the directory **dimension** implements the grid-related issues. **coefficient** implements the coefficient object and the functions to initialize it by diagonalization. **observable** deals with the evaluation of dipolar observables. **operator** prepares the matrix element data required in propagation. **propagator** implements all propagators used in QPC-TDSE. **spectrum** provides function calls to extract momentum distributions, projections, wave-function values etc. **utilities** are some fundamental operations required by the other components. Besides, the header source **potential.hpp** contains the predefined potentials, and **absorber.hpp** just work as its name.