Creation of a Graphical User Interface for the Analysis of Quantum Dynamics Simulations

Wenzhao Jia

A dissertation presented for the degree of
Master of Science
of
University College London

Scientific and Data Intensive Computing
Department of Physics and Astronomy
September 2023

Abstract

The *Quantics* package is a set of programs for quantum dynamics simulations that solves the time-dependent Schrödinger equation for a set of initial conditions defined by the user via input files. The propagated wavefunction, as well as other details from the calculation, can then be analysed using *Quantics*' analysis programs, a library of approximately 60 standalone executables that are called on a command line interface (CLI). The aim of this dissertation is to detail to development of a graphical user interface (GUI) which bundles the most important analysis programs together into one program coupled with a user-friendly interface that allows the user to select, execute, and visualise analysis programs without the cumbersome and manual use of a CLI, as well as to produce a solid framework such that integration with future analysis programs are as easy as possible.

Disclaimer

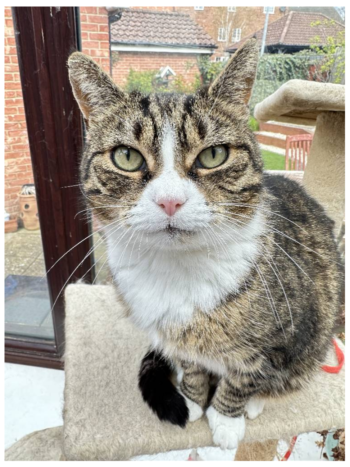
I, Wenzhao Jia, confirm that the work presented in this dissertation is my own. Where information has been derived from other sources, I confirm that this has been indicated in the dissertation.

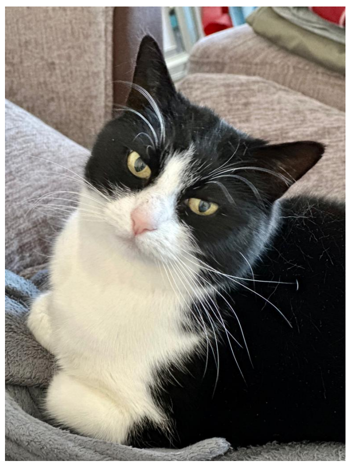
The accompanying code for this dissertation can be found in the `analysis_gui` branch of the *Quantics* GitLab repository, available at `https://gitlab.com/quantics/quantics/-/tree/analysis_gui`. Access to the repository is granted on request by contacting Graham Worth at `g.a.worth@ucl.ac.uk`.

Acknowledgements

For supervision over this project, I would like to thank Graham Worth and especially Michael Parkes for the weekly meetings which helped encourage me to keep me on track by allowing me to report new implementations in the code to him every week. His support during the write-up of the dissertation was also greatly appreciated. Overall, the dissertation ended up with almost 30,000 words; for comparison the recommended word count around 8,000. I definitely overextended with the project!

Secondly, I would like to give my thanks to programme tutor Stephen Fossey and academic advisor Joseph Frost-Schenk for staying in touch with me over the late summer period through email. Finally, I must thank my parents, grandparents, and our two family cats who have been present alongside me through some dark times.


Murphy.


Domino.

Contents

1.1 The Quantics package

Molecular quantum dynamics deals with the behaviour of molecular systems. Such behaviour consists of the motion of nuclei and electrons, whose properties are governed by quantum mechanics. The realm of quantum mechanics give rise to various chemical and physical phenomena such as molecular interactions, electron scattering, and bond breaking, which can be studied via the movement of these particles.[1]

These phenomena occur at the *femtosecond* time scale. The field of study around this time-scale is called *femtochemistry*,[2] and experiments that operate at the atomic and ultra-fast level require accompanying calculations to obtain a molecular interpretation. These calculations can be performed by quantum dynamics simulations.

The *Quantics* package is a suite of programs dedicated to such a task.[3] By solving the time-dependent Schrödinger equation, the time-evolution of the molecular system can be examined, and complete information on quantum effects can be observed. The origins of the *Quantics* package stem from when Uwe Manthe, as part of his PhD, created the initial code[4] in Fortran, which is the primary programming language used for the package to this day.[5]

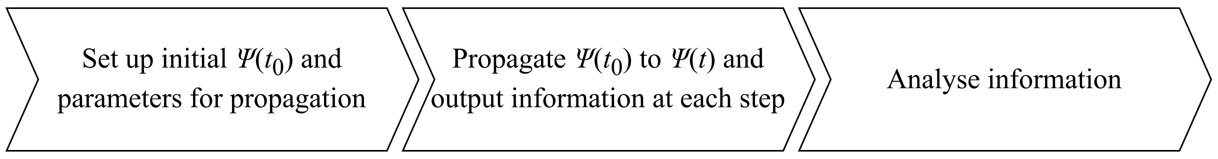

Set up initial $\Psi(t_0)$ and parameters for propagation
Propagate $\Psi(t_0)$ to $\Psi(t)$ and output information at each step
Analyse information

Figure 1: The three underlying processes of a quantum dynamics simulation. Ψ is the wavefunction, which describes the quantum system in its entirety.

A quantum dynamics simulation can be divided into three parts:[6]

1. **Setting up the initial conditions for the simulation.** For the *Quantics* package, this will involve writing an input file. A previous SDIC student has written a tentative GUI for this purpose. Other files may also be required, such as an operator file for the Hamiltonian and a database for direct dynamics calculations. For the purposes of this dissertation, excessive detail for this part will not be given, but an example can be seen in §1.2.
2. **Simulation of the dynamics performed by solving the time-dependent Schrödinger equation.** The *Quantics* package includes many methods to do this, but the most commonly used are MCTDH, explained in §2.4, and DD-vMCG, described in §2.6. The method to use is specified in the input file. The user can then call `quantics` on the command line interface to perform the simulation, specifying the relative location of the input file.
3. **Analysis of the output of the simulation.** This is the focus point of the dissertation. A library of approximately 60 standalone programs are included in the *Quantics* package which can read the various outputs of the simulation. These programs are also called on the command line, but each program has a different set of optional parameters and inputs, which can become rather unwieldy.

In order to showcase these steps, an example of a quantum dynamics simulation using *Quantics* is given in the next section. To compare and contrast, we will use the original command line programs to execute analyses.

1.2 An example with butatriene

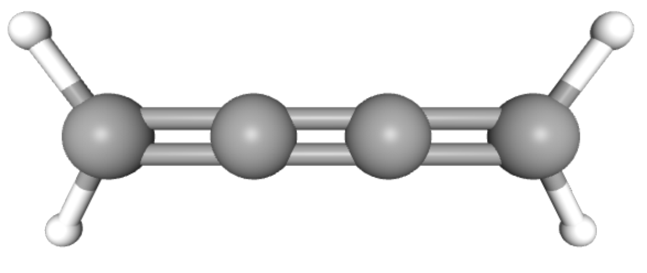


Figure 2: Molecular structure of butatriene. Grey atoms are carbon, white are hydrogen.

The butatriene cation is a molecule with chemical formula $C_4H_4^+$ (Fig. 2). We wish to investigate the *photoionisation* of butatriene, which can be done by plotting its ionisation or absorption spectra. This can be thought of as the energy required to "excite" the ground state S_0 to one of two excited states S_1 and S_2 (Fig. 3).

For our propagation method, we will use MCTDH. To start a *Quantics* simulation we first need to specify the Hamiltonian of the system. Here, a vibronic coupling Hamiltonian is used, in the form

$$\hat{H} = \sum_{i=1}^{18} \frac{\omega_i}{2}\left(\frac{\partial^2}{\partial Q_i^2} + Q_i^2\right)\mathbb{1} + \sum_{i\in G_1}\begin{bmatrix}\kappa_i^{(1)} & 0\\ 0 & \kappa_i^{(2)}\end{bmatrix} Q_i + \sum_{i\in G_3}\begin{bmatrix}0 & \lambda_i\\ \lambda_i & 0\end{bmatrix} Q_i + \text{higher order terms} \qquad (*)$$

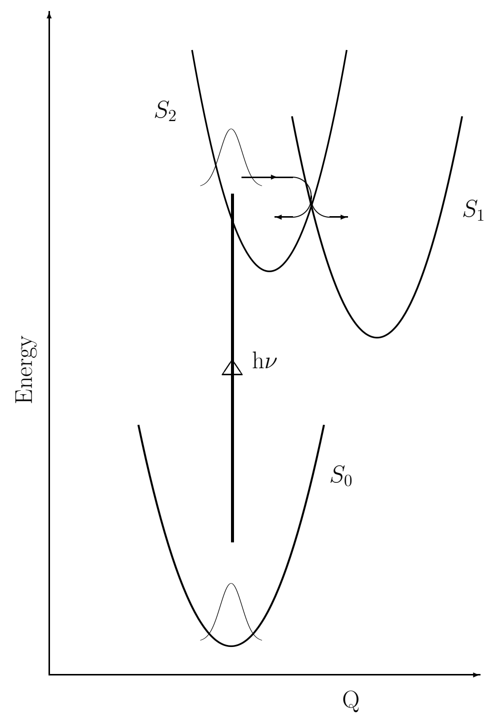


Figure 3: Schematic diagram of excitation in butatriene.

The Hamiltonian is defined in an operator file, where the form of the Hamiltonian is given as well as the values of its various parameters. One of the parameters is `init_state`, which can place the wavepacket on the upper or lower excited state. We need to run calculations for both states to obtain the full spectrum. Then, we need to specify details such as the propagation time, the initial wavefunction as well as technical details for MCTDH such as the number of points on the grid for each degree of freedom, which is included in an input file. The contents of the operator and input files are in a particular format that *Quantics* can read. For the interested reader, see appendix §8.2 for the full listing.

If we have named our input file `butatriene.inp`, we simply need to type on the command line `quantics -mnd butatriene.inp` to perfrom the simulation. The `-mnd` flag sets the directory name with the output of the simulation the same as the name of the input. If we navigate to the newly created directory, we will see the output files from the simulation:

```
auto      check     dvr       gridpop   input
log       op.log    oper      output    psi
restart   scratch   speed     timing    update
```

Note that different files may be generated depending on parameters chosen for the propagation. We can then perform the analysis on the command line. To compute the absorption spectra we use the `autospec` command. If we do not know the parameters of the command, we can first invoke the command with the help flag, using `autospec -h`. We get the following output:

```
Purpose: Computation of the spectrum by fourier-transforming
         the autocorrelation function.
         Three spectra are generated acording to the weight functions
         cos^n(pi*t/(2*T)) (n=0,1,2). The option -lin allows to use
         a second set of filters. In plspec the filters of this
         second set are enabled through the option -g3, -g4, or -g5.
         Additionally there is the weight function exp(-(t/tau)^iexp).
         The conversion factor from damping time to FWHM energy width
         is 1.32 eV fs and 2.2 eV fs for iexp=1 or 2, respectively.
         Input a zero for tau if not damping is required.

Usage : autospec [-f -i -o -e -n -r -ctr -ph -p -q -g -t -FT -EP -Mb -ver -h -? -inter]
                [emin emax unit tau iexp]

Options :
-f FILE : The autocorrelation is read from file FILE rather than from ./auto
           The string FILE may be a relative or full path-name.
-i DIR  : Take DIR as input directory.
...
```

Listing 1: Snippet of the help text for the `autospec` program. The output is continued by further explanation of all of the parameters.

Every analysis program has an `-h` flag for this purpose. In this scenario, we would like to see the spectra between 9.0 and 10.5 eV, with a damping time of 30 fs at the first order. We can specify this only using the positional parameters `autospec 9.0 10.5 ev 30 1`. This creates another file named `spectrum.pl`, which can be read using *gnuplot* using the `gnuplot` command.

gnuplot is also a program the functions on the command line. We pass it the *gnuplot*-specific command `plot 'spectrum.pl' u 1:2 w l`, which is to create a line plot of the first column, energy, against the second, intensity. There are further columns for the intensity for a different variant of the formula, but we will use the one with the g_0 damping function. This will show a popup with the resulting plot (Fig. 4). We then repeat the simulation changing the parameter `init_state = 2` to `1` in the input file to simulate the excitation onto the other state.

Having obtained two `spectrum.pl` files we need to add their contents together. Firstly, they must be in the same folder, which requires copying either using the `cp` command or in a file explorer. Then, we paste the contents of both files together using the command `paste spectrum1.pl spectrum2.pl > spectrumc.pl`. To view this using *gnuplot*, the command `plot 'spectrumc.pl' u 1:($2+$6) w l` must be used to add the second and sixth column together before plotting[a].

Figure 5 shows the simulated spectra that was performed with *Quantics* against the spectrum that was obtained via experiment. We see that the MCTDH calculation aligns very closely with experimental laboratory data! Note that the files given in §8.2 and shown in Fig. 4 are for the linear terms only (equation ($*$)), while Fig. 5 uses the spectrum calculated up to quadratic terms. They are similar, but the higher-order model is slightly more accurate.

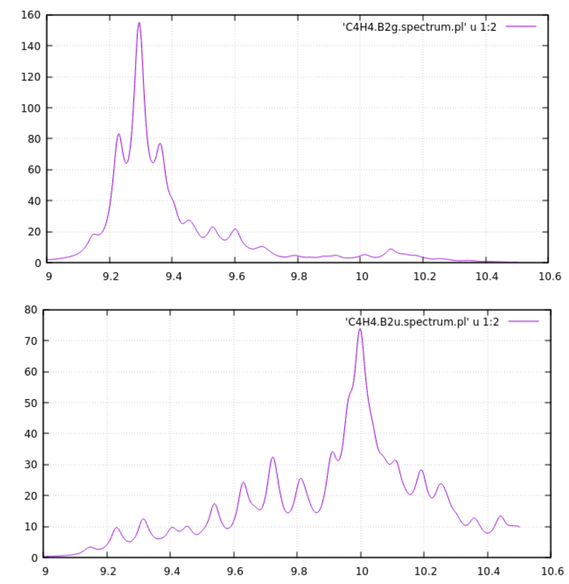


Figure 4: Partial spectra of excitation to each of the two excited states shown in *gnuplot*. They must be adding to obtain the combined spectrum in Fig. 5.

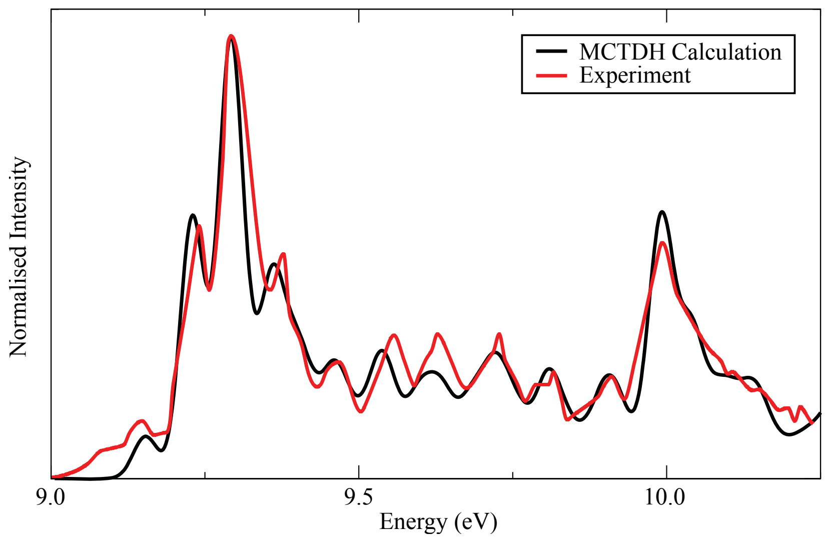


Figure 5: Calculated absorption spectrum of butatriene (quadratic model) in black,[7] and the spectrum from an experiment.[8].

For this particular molecule, the two peaks at around 9.3 eV and 10 eV have a simple interpretation; they are from the populations of the lower ionic state and the higher ionic state, respectively.[9] The intensities in the middle however, are due to *vibronic states* that act as a mixture of the two states. These are formed as a result of a *conical intersection* between the potential energy surfaces of the two excited states[10] (see §2, Fig. 8).

1.3 Aims and objectives

As seen in the previous section, the use of typing in commands in the command line window is very inconvenient. If we focus on the tasks required to plot the absorption spectra, we see that the user needed to know:

- The name of the required *Quantics* analysis program, namely `autospec`;
- The inputs of the analysis program, which can be seen by using the help flag `-h`;
- The form of the output of the analysis program, namely that for the `spectrum.pl` file the first column in energy and the second is intensity (for the damping function g_0);
- The gnuplot commands that are required to plot the values in the columns of the output.

It is clear that to performing this one analysis as a person unfamiliar with any of these tasks would be very challenging. In the past, there has been some steps in making the analysis more user-friendly.

- There are a number of shell scripts, or 'pl'-scripts that automatically plot the output of the analysis program in *gnuplot*. The script still requires the input parameters so it can call the corresponding analysis program. There exists one called `plspec` for plotting the absorption spectra which we did not use.
- Some analysis programs, including `autospec`, come with an `-inter` flag (for "interactive"), which replaces some of these parameters with a command line menu (Listing 2). These menus come as default for some other programs, e.g. `showsys` which plots the potential energy surface (PES) or the time-evolution of the wavepacket, which may also have optional parameters, but in general must be interacted with using the command line input.

```
0 = stop,  q = stop,  quit = stop
1 = plot to screen
5 = save data to an xyz file
9 = toggle re-plot (get new set of contours)

10 = change plot task (plot diabatic reduced density)
```

[a]A program named `sumspec` exists in the *Quantics* package which can also be used to sum columns together. However, this program also has many parameters that the user needs to be familiar with.

```
20 = change coordinate section (  Q_5=x  Q_14=y)
30 = change coordinate bounds
40 = show coordinate info
50 = change a single coordinate
60 = change state selection (  1)
80 = change coordinate units
90 = change Z-axis units (au)

110 = toggle contour mode (linear)
120 = change number of contours (21)
150 = toggle grid lines (off)
160 = toggle surface (off)
170 = toggle contour lines (on)
240 = toggle key (off)
245 = toggle title (on)
280 = change time-slice format, e.g. movie(step-through)
285 = change time-step (  1)
```

Listing 2: The main command line menu of the `showsys` program. Only a subset of all the options are included.

- Finally, there is a program called `analysis`, which is a command line menu program which lists the analysis programs, and calls them (usually with `-inter`) when the user selects the option. Listing 3 shows the format of this program.

```
*************************************************************************
     THE QUANTICS WAVEPACKET PROPAGATION ANALYSIS PACKAGE
          Program Version :     2
          Release         :     1
*************************************************************************
Present directory is: {working directory}

  0 = stop
  1 = list / change directory
  2 = analyse convergence
  3 = analyse integrator
  4 = analyse results
  5 = analyse system evolution
  6 = analyse potential surface
>> 4
 *** Plot Results ***
 0 = return to main menu
 1 = Plot autocorrelation function
 2 = Plot Fourier Transform of autocorrelation function
 3 = Plot spectrum from autocorrelation function
 4 = Plot eigenvalues from matrix diagonalisation
```

Listing 3: The format of the `analysis` program. Typing 2 (or 3) from here will call `autospec -inter`.

However, we can go one step further. All the previous steps mentioned still required the use of the command line interface (CLI). However, for many people in the current day, graphical user interfaces (GUIs) are used where visual icons and buttons with labels appear on a screen that can be interacted with.[11] Consider the `analysis` program (Listing 3)—it would be much more efficient if we can click these options instead of typing their corresponding number.

The aim of the project is the creation of such a GUI to replace the older CLI that is necessary to perform analysis in *Quantics*. The GUI can present the user with the different analyses and allow them to choose the parameters of the analysis by interacting with the visuals on the screen. It should be user friendly by having a layout that is not confusing or necessitate looking up information on how to use it.

Ideally, all present analysis programs should be included as an option in the GUI. However, due to time limitations, only a subset of all programs were included, but this did include all the programs listed by the command-line `analysis` program (except the ones that did not work). Even if not all programs are included, the framework in the program should make integrating analyses as easy as possible.

A secondary aim is to create new analysis programs for *Quantics*, especially for DD-vMCG simulations, which is a comparatively newer method compared to MCTDH.Implementations can be found in §4.12. An extended objective, which is to check on the status of a *Quantics* simulation running on a remote server, was proposed, but never completed due to time limitations.

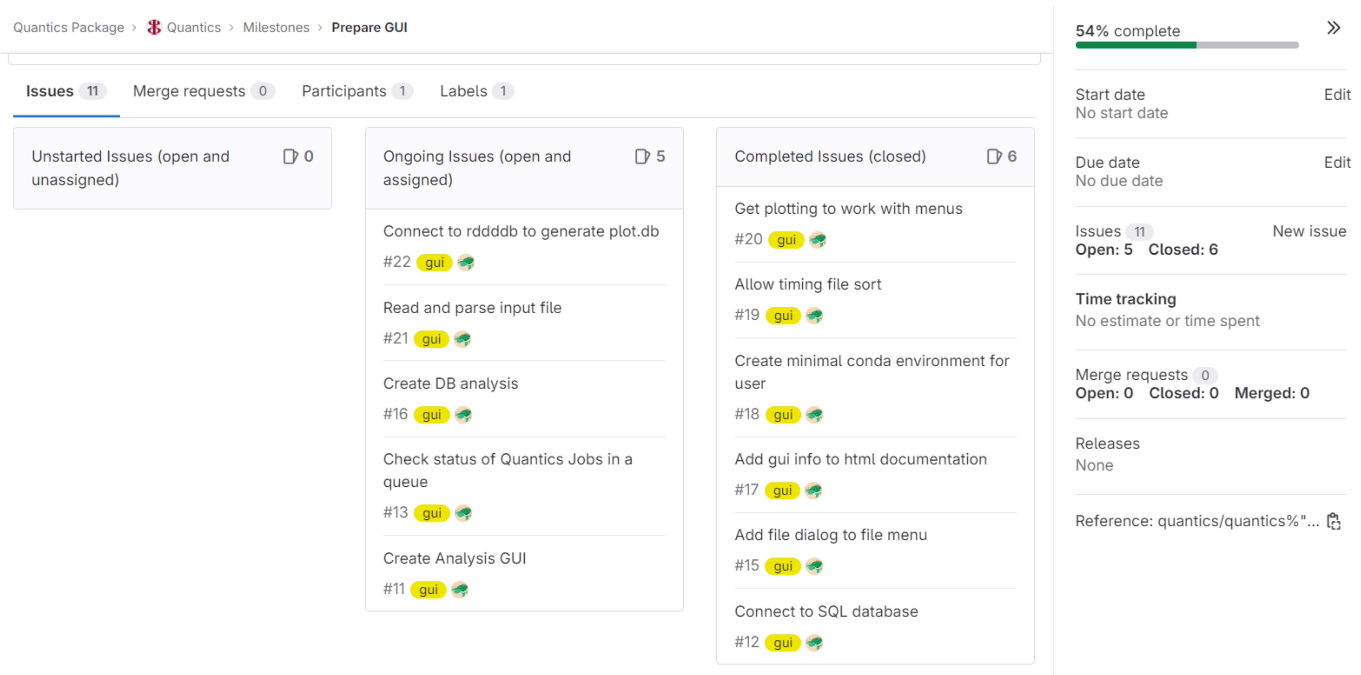
1.4 Software development basis

It is not a good idea to implement a GUI from scatch, as there are a number of packages and frameworks that already exist to help in the GUI implementation. The framework chosen is *PyQt*[b], which provides Python bindings to the underlying C++ code of Qt, a powerful cross-platform GUI framework.

One might consider why not to use the C++ interface directly, since this would be faster. The reason is because we can use *pyqtgraph* (further disscused in §4.3) to natively insert a plotting tool inside the GUI, without needing to call other programs such as *gnuplot*. *pyqtgraph* is not available in C++, so Python is therefore the programming language used for this project.

PyQt has advantages over other GUI frameworks due to its size and power; for example, it includes 200 widgets, which are representations of an item on a screen.[12] There is also a dedicated program called *Qt Designer*, which helps programmers to design a UI visually as seen by the user (§3.2). *Qt* itself has been used for software such as VLC media player[c], OBS, for screencasting and recordings[d], and is also the underlying basis of *Spyder*[e], the integrated development environment (IDE) tailored for scientific computing that I use to write Python code.

The code development of the project uses *Git*, a revision control software which allows previous versions of the program to be easily accessed, and newly introduced code can be reverted if it breaks the program.[13] Each new change can be added to a "commit" which is recorded as a snapshot of the program, and the history of commits (the log) can be used to see the history of what has been implemented.



Quantics Package › Quantics › Milestones › Prepare GUI

Issues 11 | Merge requests 0 | Participants 1 | Labels 1

Unstarted Issues (open and unassigned) 0

Ongoing Issues (open and assigned) 5
- Connect to rddddb to generate plot.db #22 gui
- Read and parse input file #21 gui
- Create DB analysis #16 gui
- Check status of Quantics Jobs in a queue #13 gui
- Create Analysis GUI #11 gui

Completed Issues (closed) 6
- Get plotting to work with menus #20 gui
- Allow timing file sort #19 gui
- Create minimal conda environment for user #18 gui
- Add gui info to html documentation #17 gui
- Add file dialog to file menu #15 gui
- Connect to SQL database #12 gui

54% complete

Start date — No start date — Edit

Due date — No due date — Edit

Issues 11 — New issue
Open: 5 Closed: 6

Time tracking
No estimate or time spent

Merge requests 0
Open: 0 Closed: 0 Merged: 0

Releases
None

Reference: quantics/quantics%"...

Figure 6: The *GitLab milestone*, which consists of tasks set out to complete, for this project.

Git is also related to *GitLab*, a hosting service for software projects, which allows collaborative work between different authors, with features such as *milestones* to distribute tasks to different people (for this project, all were to me). A centralised repository of code can be hosted on *GitLab*, which can then be downloaded locally on a computer. Using *Git*, code can then by comitted and "pushed" back onto the online repository, and changes can be seen by other people. Code can also be committed onto different "branches", which allows parallel, non-linear development of the code, which can be merged together if necessary. A branch named `analysis_gui` is dedicated to the code created in this project.

1.5 Dissertation layout

In §2, methods for solving the time-dependent Schrödinger equation in *Quantics* are discussed. The design of the GUI and base structure of the code are explained in §3. §4 is dedicated to the implementation of individual features and analysis of the GUI. The documentation and testing of the code are described in §5. The conclusion

[b]`https://riverbankcomputing.com/software/pyqt`
[c]`https://code.videolan.org/videolan/vlc`
[d]`https://github.com/obsproject/obs-studio`
[e]`https://github.com/spyder-ide/spyder`

§6 will then detail a summary of how the aims of the project are met, any limitations to the work and how the code can be improved upon in the future.

2 ***Background***

2.1 The Schrödinger equation

For a quantum dynamics simulation to function, it must know the equations of motion for the system. These are obtained by solving the Schrödinger equation, which is akin to Newton's second law in classical mechanics. There are two forms of this equation. In its time-dependent form, it is the time-dependent Schrödinger equation (TDSE) and reads

$$i\hbar \frac{\partial \Psi}{\partial t} = \hat{H}\Psi \tag{1}$$

while in its time-independent form (TISE), reads

$$\hat{H}\Psi_n = E_n \Psi_n. \tag{2}$$

The term $\hat{H}$ is known as the Hamiltonian operator. It represents the total energy of the system, but also governs the time evolution of quantum states.[14] If the Hamiltonian is time-dependent, for example, due to an external laser field, the time-dependent version must be used. Otherwise, both equations are equivalent, and in the case of quantum dynamics simulations, which approach to take depends on numerical efficiency.

Equation (1) represents an initial value problem (IVP), while equation (2) represents an eigenvalue problem. Mathematically, IVPs are simpler but the TDSE has one additional variable, time. Historically, the TISE was preferred, but the TDSE is now generally preferred; some reasons include better approximations, being simpler than the TISE for certain processes containing continuum functions, and only requiring the properties of the initial state, whose restriction makes the problem easier to solve.[9]

In the time-dependent scheme, the wavefunction Ψ can be decomposed into wavepackets. The solution of the TDSE generates methods to propagate these wavepackets.[15] However, the TDSE (1) cannot be analytically solved for more than two particles, similarly to the three-body problem in classical mechanics. Thus, numerical methods and approximations are used to produce solutions, and the accuracy of such methods can then be experimentally verified in a laboratory.

One such approximation is known as the Born-Oppenheimer approximation, which operates in what is known as the *adiabatic* picture. It operates from an assumption that the nuclear and electronic motions of the atom can be seperated. This leads to the idea of a molecular system moving over a potential energy surface (PES), to which the analysis allows many quantum features to be described—for example, the minima represent stable configurations and saddle points represent transition states connecting these configurations.[16]

A method that solves the TDSE, known as the multiconfiguration time-dependent Hartree (MCTDH) method, is the core method of the *Quantics* package. It is designed to tackle multidimensional problems with many degrees of freedom and still produce accurate results in line with experimental data (e.g. Figure 5). The following sections detail the mathematical description of MCTDH, as well as some of its applications. §2.2 and §2.3 are predecessors to MCTDH as well as limits to MCTDH given certain circumstances. The mathematics of MCTDH are then given in §2.4. An overview of ML-MCTDH, an extension to MCTDH, is given in §2.5. Finally, the *Quantics* package also implements the direct dynamics variational multi-configurational Gaussian method (DD-vMCG), which calculates potential surfaces on the fly. This is briefly discussed in §2.6.

2.2 The standard method

In these following sections, atomic units are used where $\hbar = 1$. The time-dependent Schrödinger equation previously introduced in (1) thus reads

$$i\frac{\partial \Psi}{\partial t} = i\dot{\Psi} = \hat{H}\Psi. \tag{3}$$

A direct way to solve equation (3), known as the "standard method", is to expand the f-dimensional wavefunction Ψ into a product of time-independent basis sets per

$$\Psi(q_1, \dots, q_f; t) = \sum_{j_1=1}^{N_1} \cdots \sum_{j_f=1}^{N_f} C_{j_1,\dots,j_f}(t) \prod_{k=1}^{f} \chi_{j_k}^{(k)}(q_k) \tag{4}$$

where f is the number of degrees of freedom, $q_1, \dots, q_f$ are the nuclear coordinates, $C_{j_1,\dots,j_f}(t)$ are the expansion coefficients, $\chi_{j_k}^{(k)}$ are the basis functions for DOF k and N_k is the number of basis functions employed for DOF k.[6]

The equations of motion for the system can be derived using the Dirac-Frenkel variational principle,[17] which reads

$$\langle \delta\Psi | \hat{H} | \Psi \rangle - i \langle \delta\Psi | \dot{\Psi} \rangle = 0 \tag{5}$$

where we have

$$\delta\Psi = \sum_{l_1,\dots,l_f} \frac{\partial \Psi}{\partial C_{l_1,\dots,l_f}} \delta C_{l_1,\dots,l_f} = \sum_{l_1,\dots,l_f} \prod_{k=1}^{f} \left[\chi_{l_k}^{(k)}(q_k) \right] \delta C_{l_1 \cdots l_f}, \tag{6}$$

$$\dot{\Psi} = \sum_{j_1,\dots,j_f} \dot{C}_{j_1,\dots,j_f}(t) \prod_{k=1}^{f} \chi_{j_k}^{(k)}(q_k). \tag{7}$$

Since variations are independent,

$$\delta C_{l_1,\dots,l_f} = \begin{cases} 1 & \text{for } l_1,\dots,l_f = l_1^{(0)},\dots l_f^{(0)} \\ 0 & \text{otherwise} \end{cases}$$

which after replacing $l_k^{(0)}$ with l_k and applying (5), then results in the equations of motion for $C_{j_1,\dots,j_f}(t)$:

$$i\dot{C}_{j_1,\dots,j_f} = \sum_{l_1,\cdots,l_f} \langle \chi_{j_1}^{(1)} \cdots \chi_{j_f}^{(f)} | H | \chi_{l_1}^{(1)} \cdots \chi_{l_f}^{(f)} \rangle C_{l_1,\cdots,l_f} \tag{8}$$

which is a first-order differential equation with constant coefficients in the form[18, chap. 8.1]

$$\boldsymbol{C}(t) = e^{-i\mathbf{H}t} \boldsymbol{C}(0). \tag{9}$$

This method is numerically exact. However, it also scales exponentially with the number of degrees of freedom: for N basis points and f degrees of freedom, the computational cost is of the order N^f, which provides an effective limit of four-atom systems using this method.[5]

2.3 The time-dependent Hartree method

One of the alternatives to the standard method in §2.2 is the time-dependent Hartree method,[19, 20] where the wavefunction is expressed as a product of 1D single-particle functions (SPFs) φ_k,

$$\Psi(q_1, \dots q_f; t) = a(t) \prod_{k=1}^{f} \varphi_k(q_k, t). \tag{10}$$

The product of functions, known as a *Hartree product*, is now time-dependent as opposed to the time-independence in the standard method. $a(t)$ is a redundant term which allows us to freely choose the phases of the SPFs, and SPFs usually take the form in a time-independent basis set of primitive basis functions $\chi_{i_k}^{(k)}(q_k)$

$$\varphi_{j_k}^{(k)}(q_k, t) = \sum_{i_k=1}^{N_k} c_{i_k}^{(k,j_k)}(t) \chi_{i_k}^{(k)}(q_k). \tag{11}$$

The representation of Ψ in equation (10) is not unique—take for example, $\varphi_1 \cdot \varphi_2 = (\varphi_1/b)(\varphi_2 \cdot b)$ for any complex number $b \neq 0$. Thus we introduce constraints

$$i \langle \varphi_k(t) | \dot{\varphi}_k(t) \rangle = g_k(t) \tag{12}$$

for an arbitary real function $g_k(t)$. The constraint also ensures the norm of the SPFs are conserved since

$$\frac{\mathrm{d}}{\mathrm{d}t} ||\varphi_k||^2 = \langle \dot{\varphi}_k | \varphi_k \rangle + \langle \varphi_k | \dot{\varphi}_k \rangle = 2 \operatorname{Im} g_k = 0. \tag{13}$$

To prepare for the variation, and for convenience, definining

$$\Phi := \prod_{k=1}^{f} \varphi_k, \qquad \Phi^{(k)} := \prod_{l=1, l\neq k}^{f} \varphi_l$$

we have

$$\dot{\Psi} = \dot{a}(t)\Phi + a \sum_{k=1}^{f} \dot{\varphi}_k \Phi^{(k)}, \tag{14}$$

$$\delta\Psi = (\delta a)\Phi + a \sum_{k=1}^{f} (\delta\varphi_k) \Phi^{(k)}. \tag{15}$$

Applying (5) to the above, we obtain terms with δa and terms with $\delta\varphi_k$, which must equal 0 independently since they are independent of each other:

$$\langle(\delta a)\Phi|\,\hat{H}\,|a\Phi\rangle - i\langle(\delta a)\Phi\mid \dot{a}\Phi + a\sum_k \dot{\varphi}_k\Phi^{(k)}\rangle = 0, \tag{16}$$

$$\sum_{k=1}^{f}\Big[\big\langle(\delta\varphi_k)a\Phi^{(k)}\big|\,\hat{H}\,|a\Phi\rangle - i\big\langle(\delta\varphi_k)a\Phi\,\big|\,\dot{a}\Phi + a\sum_{k'}\dot{\varphi}_k\Phi^{(k')}\big\rangle\Big] = 0. \tag{17}$$

For (16), this simplifies to

$$(\delta a)^* a\,\langle\Phi|\,H\,|\Phi\rangle = i(\delta a)^*\dot{a} + i(\delta a)^* a\sum_k \big\langle\Phi\big|\dot{\varphi}_k\Phi^{(k)}\big\rangle$$

$$i\dot{a} = (E - \sum_{k=1}^{f} g_k)a \tag{18}$$

since $\big\langle\Phi\,\big|\,\dot{\varphi}_k\Phi^{(k)}\big\rangle = i\,\langle\varphi|\dot{\varphi}\rangle = g_k$ and introducing

$$E = \langle\Phi|\,\hat{H}\,|\Phi\rangle = \frac{\langle\Psi|\,\hat{H}\,|\Psi\rangle}{\langle\Psi|\Psi\rangle}. \tag{19}$$

For (17), we have the simplifications

$$\begin{aligned}
\big\langle(\delta\varphi_k)a\Phi^{(k)}\big|\,\hat{H}\,|a\Phi\rangle &= i\langle(\delta\varphi_k)a\Phi\mid\dot{a}\Phi\rangle + i\langle(\delta\varphi_k)a\Phi\mid a\sum_{k'}\dot{\varphi}_k\Phi^{(k')}\rangle\\
|a|^2\,\langle\delta\varphi_k|\,\big\langle\Phi^{(k)}\big|\,\hat{H}\,\big|\Phi^{(k)}\big\rangle\,|\varphi_k\rangle &= i\dot{a}a^*\,\langle\delta\varphi_k|\varphi_k\rangle + i|a|^2\,\langle\delta\varphi_k|\dot{\varphi}_k\rangle + i|a|^2\,\langle\delta\varphi_k|\varphi_k\rangle\sum_{k'\neq k}\langle\varphi_{k'}|\dot{\varphi}_{k'}\rangle\\
|a|^2\,\langle\delta\varphi_k|\,\hat{\mathcal{H}}^{(k)}\,|\varphi_k\rangle &= |a|^2(E - \sum_k g_k)\,\langle\delta\varphi_k|\varphi_k\rangle + i|a|^2\,\langle\delta\varphi_k|\dot{\varphi}_k\rangle + |a|^2\,\langle\delta\varphi_k|\varphi_k\rangle\sum_{k'\neq k} g_{k'}\\
i\,\langle\delta\varphi_k|\dot{\varphi}_k\rangle &= \langle\delta\varphi_k|\,\hat{\mathcal{H}}^{(k)}\,|\varphi_k\rangle - (E - g_k)\,\langle\delta\varphi_k|\varphi_k\rangle
\end{aligned} \tag{20}$$

where the *mean field* operator $\hat{\mathcal{H}}^{(k)} = \big\langle\Phi^{(k)}|\hat{H}|\Phi^{(k)}\big\rangle$ is introduced. Since $\delta\varphi_k$ is arbitrary,

$$i\dot{\varphi}_k = (\mathcal{H}^{(k)} - E + g_k)\varphi_k. \tag{21}$$

The choice of the function g_k can be chosen to be any real function; for Hermitian Hamiltonians one may choose $g_k = E$ or $g_k = E/f$ but the simplest choice is $g_k = 0$ in which the equations of motion become

$$\begin{cases} a(t) = a(0)e^{-i\int_0^t E(t')\,\mathrm{d}t'}\\ i\dot{\varphi}_k = (1 - |\varphi_k\rangle\,\langle\varphi_k|)\hat{\mathcal{H}}^{(k)}\varphi_k \end{cases} \tag{22}$$

where $|\varphi_k\rangle\,\langle\varphi_k|$ denote the projector onto state φ_k. The form of the wavefunction in (10) simplifies one f-dimensional differential equation into f 1-dimensional differential equations, meaning the computational cost is reduced from N^f in the standard method to a mere $N\cdot f$. As such, this method is approximate, only being exact if the Hamiltonian is seperable, i.e. $V_{\text{corr}} = 0$ in

$$\hat{H} = \sum_{k=1}^{f} h^{(k)} + V_{\text{corr}} \tag{23}$$

where $h^{(k)}$ only operates on DOF k. To investigate the error, consider the TDH as the solution to the TDSE with an effective Hamiltonian

$$i\dot{\Psi} = \hat{H}_{\text{eff}}\Psi. \tag{24}$$

It can be shown that

$$\hat{H} - \hat{H}_{\text{eff}} = V_{\text{corr}} - E_{\text{corr}} - \sum_{k=1}^{f}\Big(\langle\prod_{k'\neq k}\varphi_{k'}|\,V_{\text{corr}}\,|\prod_{k'\neq k}\varphi_{k'}\rangle - E_{\text{corr}}\Big) \tag{25}$$

where $E_{\text{corr}} = {}^{\langle\Psi|V_{\text{corr}}|\Psi\rangle}/_{\langle\Psi|\Psi\rangle}$.[18, chap. 8.2] The method has been used in the simulations of many-atom systems consisting of ≈ 100 modes, involving phenomena such as photodissociation and electron photodetachment.[21] However, due to the lack of correlation between the DOFs, the TDH approach is not suitable for the majority of applications, such as the $H + H_2$ molecular system,[22] as several configurations are required. This naturally leads to the multiconfiguration time-dependent Hartree method (MCTDH) in §2.4.

2.4 The multiconfiguration time-dependent Hartree method

In order to recover some the missing correlation in TDH, we turn to using multiple configurations. First, we have the multiconfigurational wave function in the form[4]

$$\Psi(q_1,\dots q_f;t) = \sum_{j_1=1}^{n_1}\cdots\sum_{j_f=1}^{n_f} A_{j_1,\dots,j_f}(t)\prod_{k=1}^{f}\varphi_{j_k}^{(k)}(q_k,t) \tag{26}$$

where the SPFs are in the same form as in TDH (11). Both TDH and the standard method are limits to this wavefunction. If $n_i = 1$ for $i = 1,\dots,f$, this is equivalent to TDH, and if $n_i = N_i$ for $i = 1,\dots,f$ then this is equivalent to the standard method. MCTDH can therefore be considered to fully cover the range of approximations from TDH to an numerically exact solution as n_i is increased from 1 to N_i.[6] Like in the TDH approach, (26) is not a unique representation as one can perform a linear transformation on the SPFs then perform its inverse transformation onto A. The constraints here are

$$i\left\langle\varphi_j^{(k)}(t)\middle|\dot\varphi_l^{(k)}(t)\right\rangle = \left\langle\varphi_j^{(k)}(t)\middle| g^{(k)}\middle|\varphi_l^{(k)}(t)\right\rangle \qquad \left\langle\varphi_j^{(k)}(0)\middle|\varphi_l^{(k)}(0)\right\rangle = \delta_{jl}. \tag{27}$$

It is assumed that $g^{(k)} = 0$ in this section for simplicity, but it is possible to derive the MCTDH equations of motion for a general $g^{(k)} \neq 0$ with more algebra.[18, chap. 8.8] For convenience, we define composite indices as

$$\begin{aligned} J &:= j_1,\dots,j_f \\ J^k &:= j_1,\dots,j_{k-1},j_{k+1},\dots,j_f \\ J_l^k &:= j_1,\dots j_{k-1},l,j_{k+1},\dots,j_f \end{aligned}$$

and configurations as

$$\Phi_J := \prod_{k=1}^{f}\varphi_{j_k}^{(k)} \qquad \Phi_{J^k} := \prod_{k'=1,k'\neq k}^{f}\varphi_{j_{k'}}^{(k')}.$$

This allows us to define *single-hole functions*, which represent the product of SPFs that do not contain the SPF for DOF k, as

$$\Psi_l^{(k)} = \sum_{J^k} A_{J_l^k}\Phi_{J^k} \tag{28}$$

thus allowing the MCTDH wavefunction (26) to be written as

$$\Psi = \sum_J A_J\Phi_J = \sum_l \varphi_l^{(k)}\Psi_l^{(k)}. \tag{29}$$

for any $k = 1,\dots,f$, as well as the mean fields

$$\langle\hat H\rangle_{jl}^{(k)} = \left\langle\Psi_j^{(k)}\middle|\hat H\middle|\Psi_l^{(k)}\right\rangle \tag{30}$$

and *density matrices*

$$\rho_{jl}^{(k)} = \left\langle\Psi_j^{(k)}\middle|\Psi_l^{(k)}\right\rangle = \sum_{J^k} A^*_{J_j^k}A_{J_l^k}. \tag{31}$$

to be defined. The density matrix is related to the density operator

$$\rho^{(k)}(q_k,q_k') = \sum_{j,l}^{n_k}\varphi_j^{(k)}(q_k)\rho_{lj}^{(k)}\varphi_l^{(k)}(q_k'), \tag{32}$$

whose diagonal values highlight how the wavepacket moves along the coordinate k. Furthermore, the diagonalisation of the density operator also gives the *natural populations* (eigenvalues) and *natural orbitals* (eigenvactors). A small natural population indicates the converge of MCTDH.[23]

The variation and time differentiation of (29) can be calculated as

$$\delta\Psi = \Phi_J\,\delta A_J = \Psi_j^{(k)}\,\delta\varphi_j^k \tag{33}$$

and

$$\dot\Psi = \sum_J \dot A_J\Phi_J + \sum_{k=1}^{f}\sum_{j=1}^{n_k}\dot\varphi_j^{(k)}\Psi_j^{(k)}. \tag{34}$$

Applying the variational principle (5) we first have the variation of A_J,

$$\begin{aligned} \langle\Phi_J|\,\hat H\,|\Psi\rangle &= i\left\langle\Phi_J\middle|\dot\Psi\right\rangle \\ \sum_L\langle\Phi_J|\,\hat H\,|\Phi_L\rangle A_L &= i\sum_L\left\langle\Phi_J\middle|\dot A_L\Phi_L\right\rangle + i\sum_k\sum_l\left\langle\Phi_J\middle|\dot\varphi_l^{(k)}\Psi_l^{(k)}\right\rangle \\ &= i\dot A_J + i\sum_k\sum_l\left\langle\varphi_{j_k}^{(k)}\middle|\dot\varphi_l^{(k)}\right\rangle\left\langle\Phi_{J^k}\middle|\Psi_l^{(k)}\right\rangle = i\dot A_J. \end{aligned} \tag{35}$$

Then comes the variation of $\varphi_j^{(k)}$,

$$\begin{aligned}
\langle \Psi_j^{(k)} | \hat{H} | \Psi \rangle &= i \sum_L \left\langle \Psi_j^{(k)} | \Phi_L \right\rangle \dot{A}_L + i \sum_L \langle \Psi_j^{(k)} | \sum_m \sum_l \dot{\varphi}_l^{(m)} \Psi_l^{(m)} \rangle \\
\langle \Psi_j^{(k)} | \hat{H} | \sum_l \Psi_l^{(k)} \varphi_l^{(k)} \rangle &= \sum_L \left\langle \Psi_j^{(k)} | \Phi_L \right\rangle \dot{A}_L \langle \Phi_L | \hat{H} | \Psi \rangle + i \langle \Psi_j^{(k)} | \sum_l \dot{\varphi}_l^{(k)} \Psi_l^{(k)} \rangle \\
\sum_l \langle \hat{H} \rangle_{jl}^k \varphi_l^{(k)} &= \hat{P}^{(k)} \sum_l \langle \hat{H} \rangle_{jl}^k \varphi_l^{(k)} + i \sum_l \rho_{jl}^{(k)} \dot{\varphi}_l^{(k)} \\
i \dot{\varphi}_j^{(k)} &= \sum_{m,l} (\rho_{jm}^{(k)})^{-1} (1 - \hat{P}^{(k)}) \langle \hat{H} \rangle_{ml}^k \varphi_l^{(k)} \qquad (36)
\end{aligned}$$

where we defined the MCTDH projector as

$$\hat{P}^{(k)} = \sum_{j=1}^{n_k} \left| \varphi_j^{(k)} \right\rangle \left\langle \varphi_j^{(k)} \right| . \qquad (37)$$

If we also define vectors of SPFs as

$$\boldsymbol{\varphi}^{(k)} = (\varphi_1^{(k)}, \dots, \varphi_{n_k}^{(k)})^\mathsf{T} \qquad (38)$$

then the MCTDH equations of motion for $g^{(k)} = 0$ become[18, chap. 8.3]

$$\begin{cases} i \dot{A}_J = \sum_L \langle \Phi_J | \hat{H} | \Phi_L \rangle A_L \\ i \dot{\boldsymbol{\varphi}}^{(k)} = (1 - \hat{P}^{(k)}) (\boldsymbol{\rho}^{(k)})^{-1} \langle \mathbf{H} \rangle^{(k)} \boldsymbol{\varphi}^{(k)}. \end{cases} \qquad (39)$$

These equations are in the form of a coupled non-linear differential equation, which can be solved using a numerical integration scheme such as Runge–Kutta, or a specialised scheme known as constant mean field.[24] MCTDH has been used to investigate many phenomena such as photodissociation processes (e.g. in NOCl[25], NO_2[26]), molecule surface scattering (e.g. for H_2/LiF[27]), the hydrogen exchange reaction $H + H_2 \longrightarrow H_2 + H$,[28] and the photo-excitation of pyrazine, whose calculation included 24 nuclear degrees of freedom.[29]

The computational effort for MCTDH generally scales per $fnN + n^f$, where N is the grid length and n the number of SPFs. As such, it still suffers from exponential scaling, but the base of the exponent can be reduced in certain ways. The following section, §2.5, details mode combination and multilayer MCTDH, which aim to reduce the computational effort.

2.5 Mode combination and multilayer MCTDH

The SPFs introduced in (11) do not need to depend on a single DOF. By combining coordinates as

$$Q_k = (q_{k,1}, q_{k,2}, \dots, q_{k,d}) \qquad (40)$$

we introduce *logical coordinates* (also known as "particles"), where the k^{th} logical coordinate depends on d physical coordinates, such that the SPFs can be written as

$$\varphi_{j_k}^{(k)}(Q_k, t) = \sum_{i_1=1} \cdots \sum_{i_d=1} C_{i_1,\dots,i_d}^{(k,j_k)}(t) \chi^{(k,1)}(q_{k,1}) \cdots \chi^{(k,d)}(q_{k,d}). \qquad (41)$$

The wavefunction is still identical to (26) except it now runs over the logical coordinates up to p instead over the total number of DOFs f. This is known as *mode combination.*

Using this method, MCTDH's A-vector can be made considerably smaller but the SPFs being multidimensional means they are harder to propagate. In general, only DOFs with strong coupling should be combined, as "over-combining" means propagation of the SPFs will take much longer than the propagation of the A-vector. The computational effort with mode combination is $p\tilde{n}N^d + \tilde{n}^p$, where we see the number of DOFs f changed to the number of mode combinations p and since the grid is now multidimensional, $N \to N^d$. If $\tilde{n} = n^d$, the A-vector length stays identical and nothing is gained. As a rule of thumb, picking $\tilde{n}$ such that $\tilde{n} \approx d \cdot n$ gives a good balance.[9]

However, there is a technique to propagate multidimensional wavefunctions efficiently which we already know: MCTDH. Instead of using MCTDH to only propagate the wavefunction, we propagate the SPFs using MCTDH as well. This technique is known as multilayer MCTDH, or ML-MCTDH.[30] Here, the wavefunction is similar to normal MCTDH,

$$\Psi(Q_1^1, \dots, Q_p^1; t) = \sum_{j_1=1}^{n_1} \cdots \sum_{j_p=1}^{n_p} A_{j_1,\dots,j_p}^1(t) \prod_{k=1}^{p} \varphi_{j_k}^{(1:k)}(Q_k^1, t) \qquad (42)$$

where the superscript 1 refers to the top layer of SPFs, but the SPFs recursively depend on the next layer of SPFs,

$$\varphi_m^{(l:k)}(Q_k^l,t) = \sum_{i_1=1} \cdots \sum_{i_{d_k}=1} A^{l+1:k}_{m:i_1,\ldots,i_{d_k}}(t)\varphi_{i_1}^{(l+1:k,1)}(Q_1^{l+1:k},t)\cdots\varphi_{i_{d_k}}^{(l+1:k,d_k)}(Q_{d_k}^{l+1:k},t) \tag{43}$$

where l refers to the layer and at the last layer the SPFs are the basis functions instead.[31] There is now an equation of motion in each layer, with the form of projector, density matrix, and mean field now depending on the layer. They have a similar form to the normal MCTDH equations of motion (39), but the calculation of the density matrices and mean-fields are more involved, since they are also calculated recursively.[32]

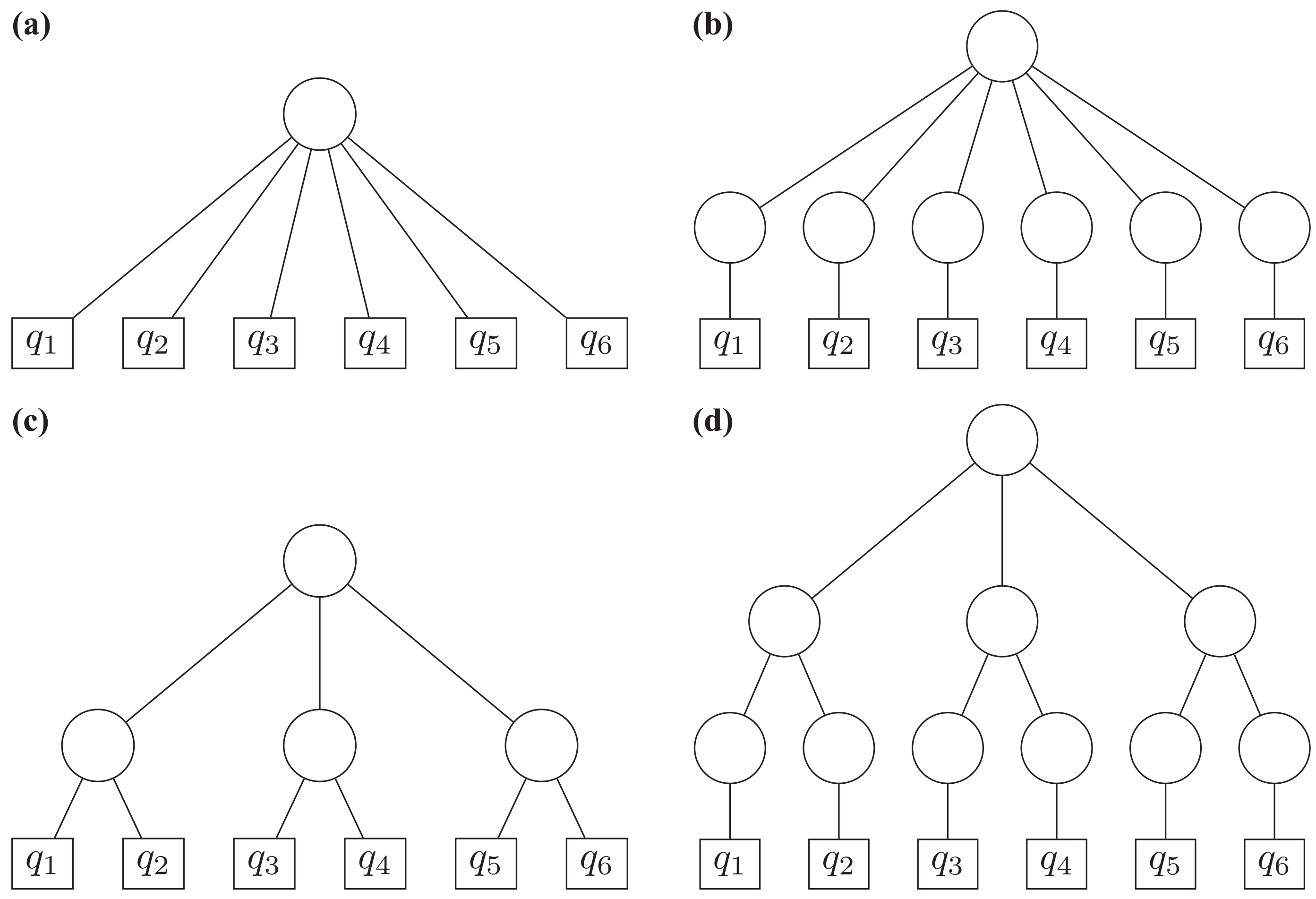


Figure 7: Wavefunctions expanded as tree structures, where circles represent a set of coefficients (A-vectors in MCTDH) and squares represent basis functions. (a) Standard method. (b) MCTDH. (c) MCTDH with mode combination. (d) Example of a 3-layer ML-MCTDH structure.[9]

Mode combination and ML-MCTDH can be represented as a tree diagram, seen in Fig. 7. In subfigure (a), we have the standard method wavefunction, which is a product of the primitive basis functions, and so are directly connected to the expansion coefficients (C in equation (4)). In subfigure (b), we have the MCTDH wavefunction, which is a product of SPFs, each of which are connected to the basis functions. In subfigure (c), mode combination is used to combine two DOFs into one SPF. In subfigure (d), the wavefunction is connected three SPFs, each of which also depend on the 2 SPFs below them. Since there are three rows of circles, this is a 3-layer ML-MCTDH structure: this way, normal MCTDH can be seen as a 2-layer structure. Note that the ML-MCTDH tree does not need to be balanced or symmetrical as shown in the figure, can have more than 3 layers, and can be used in conjuction with mode combination.

The structure of an ML-MCTDH calculation is important, as certain structures have calculations which are more difficult to converge than others.[31] ML-MCTDH works best when there are many DOFs which are not strongly correlated, for example in system-bath models in condensed-phase environments[33, 34] which can contain thousands of degrees of freedom. It has also been used to investigate phenomena such as electron transfer in a porphyrin-quinone complex and in semiconductors.[35, 36]

2.6 The direct dynamics variational multi-configurational Gaussian method

MCTDH is known as a grid-based method, which expresses the wavefunction and Hamiltonian on a finite, pre-defined grid. As such, the computation of the PES for the grid is required prior to any calculation being performed even if the system never visits all grid points, which may become problematic for larger chemical systems.[37]

Ideally, we only want to calculate the PES without pre-fitting and only when it is required. The variational multi-configurational Gaussian (vMCG) method uses Gaussian wavepackets (GWPs) as basis functions, each of which are localised in space, such that an approximation of the PES around the wavepacket can be used. In conjuction with vMCG, *direct dynamics* allows the method to calculate the potential energy function around this point on-the-fly using a quantum chemistry program such as *Gaussian*[f] or *Molpro*[g].[38]

A precursor to the vMCG method is G-MCTDH, which replaces some of the SPFs in MCTDH with a set of parameterised basis functions $g_{j_k}^{(k)}(q_k, t)$, commonly in the form of Gaussian basis functions, and has the wavefunction

$$\Psi(q_1, \ldots q_f; t) = \sum_{j_1=1}^{n_1} \cdots \sum_{j_p=1}^{n_p} A_{j_1,\ldots,j_p}(t) \prod_{k=1}^{m} \varphi_{j_k}^{(k)}(q_k, t) \prod_{k=m+1}^{p} g_{j_k}^{(k)}(q_k, t). \tag{44}$$

This way, a more limited set of parameters is propagated and so saves memory compared to MCTDH.[39] However, the equations of motion for this method is unstable due to the non-orthogonal basis of the parameterised functions. If no grid-based SPFs are used however, one arrives at the vMCG method, with the wavefunction

$$\Psi(\boldsymbol{x}, t) = \sum_{j=1}^{n} A_j(t) g_j(\boldsymbol{x}, t) \tag{45}$$

where we have the multidimensional Gaussian functions with all DOFs combined

$$g_j(\boldsymbol{x}, t) = \exp(\boldsymbol{x}^\mathsf{T} \cdot \boldsymbol{\varsigma}_j \cdot \boldsymbol{x} + \boldsymbol{\xi}_j \cdot \boldsymbol{x} + \eta_j) \tag{46}$$

where the parameters of the equation, $\boldsymbol{\Lambda}_j = (\boldsymbol{\varsigma}_j, \boldsymbol{\xi}_j, \eta_j)$, which are a square matrix, a vector, and a scalar respectively, are generally time-dependent. The diagonals of the width matrix $\boldsymbol{\varsigma}_j$ are usually kept fixed for stability (known as "frozen" Gaussians).

The equations of motion can be derived again from the Dirac-Frenkel variational principle (5), giving

$$\delta\Psi = \delta A_j g_j = \delta \Lambda_{ja} A_j \frac{\partial g_j}{\partial \Lambda_{ja}} \tag{47}$$

where Λ_{ja} is a parameter of $\boldsymbol{\Lambda}_j$. For the variation of A_j, the resulting equation of motion is

$$i\dot{A}_j = \sum_{lm} [\mathbf{S}^{-1}]_{jl} (H_{lm} - i\tau_{lm}) A_m, \tag{48}$$

where $\mathbf{S}$ is the overlap matrix $S_{ij} = \langle g_i | g_j \rangle$, $\mathbf{H}$ is a Hamiltonian matrix $H_{ij} = \langle g_i | \hat{H} | g_j \rangle$, and $\boldsymbol{\tau}$ is the overlap time-derivative matrix $\tau_{ij} = \langle g_i | \dot{g}_j \rangle$.

For the GWP parameters, we have

$$i\dot{\boldsymbol{\Lambda}} = \mathbf{C}^{-1} \boldsymbol{Y} \tag{49}$$

where $\mathbf{C}$ and $\boldsymbol{Y}$ have the following forms:

$$C_{i\alpha, j\beta} = \rho_{ij} \left(S_{ij}^{(\alpha\beta)} - [\mathbf{S}^{(\alpha 0)} \mathbf{S}^{-1} \mathbf{S}^{(0\beta)}]_{ij} \right), \tag{50}$$

$$Y_{i\alpha} = \sum_j \rho_{ij} \left(H_{ij}^{(\alpha 0)} - [\mathbf{S}^{(\alpha 0)} \mathbf{S}^{-1} \mathbf{H}]_{ij} \right). \tag{51}$$

These also have the additional definitions[38]

$$\begin{aligned} \rho_{ij} &= A_i^* A_j, & H_{ij}^{(\alpha 0)} &= \left\langle \frac{\partial g_i}{\partial \Lambda_{i\alpha}} \middle| H \middle| g_j \right\rangle, \\ S_{ij}^{(\alpha\beta)} &= \left\langle \frac{\partial g_i}{\partial \Lambda_{i\alpha}} \middle| \frac{\partial g_j}{\partial \Lambda_{j\beta}} \right\rangle, & S_{ij}^{(\alpha 0)} &= \left\langle \frac{\partial g_i}{\partial \Lambda_{i\alpha}} \middle| g_j \right\rangle. \end{aligned} \tag{52}$$

The evaluation of the equations of motion thus requires evaluation of the Hamiltonian, which depends on the potential energy surface. A Taylor series can be used to obtain the PES around the centre of a GWP $\boldsymbol{x}_0$ in what is known as the *local harmonic approximation* (LHA),

$$V(\boldsymbol{x}) = V(\boldsymbol{x}_0) + \text{gra}(\boldsymbol{x}_0)^\mathsf{T} \cdot (\boldsymbol{x} - \boldsymbol{x}_0) + \frac{1}{2} (\boldsymbol{x} - \boldsymbol{x}_0)^\mathsf{T} \cdot \text{hes}(\boldsymbol{x}_0) \cdot (\boldsymbol{x} - \boldsymbol{x}_0) \tag{53}$$

where $\text{gra}(\boldsymbol{x}_0)$ is the gradient and $\text{hes}(\boldsymbol{x}_0)$ is the Hessian matrix. These values, along with the PES, are stored in a quantum chemistry (QC) database. At each QC calculation, these values are appended to their respective tables.

[f] `https://gaussian.com/`
[g] `https://www.molpro.net/`

However, it is expensive to perform a QC calculation at each step. Instead, calculations are only run when the GWP centre has moved significantly from the geometry at the last step, controlled by a parameter known as `dbmin`. Otherwise, *Shepard interpolation* is used to obtain the energies, gradients, and Hessians:

$$V(\boldsymbol{q}) = \sum_i \frac{\nu_i(\boldsymbol{q})}{\sum_j \nu_j(\boldsymbol{q})} T_i \tag{54}$$

where T_i is a Taylor series expansion at database entry i and

$$\nu_i(\boldsymbol{q}) = \frac{1}{|\boldsymbol{q} - \boldsymbol{q}_i|^{2p}} \tag{55}$$

in which $p = 2$ generally gives the best results.[40]

It is seen in the LHA (53) that the first and second derivatives (gradient and Hessian) of the PES need to be defined. This may not be the case in the adiabatic scheme generated by the QC program where electronic states become degenerate (have the same energy) at particular points in the molecular configuration space.[41] This is known as a *conical intersection* between the two potential energy surfaces (Fig.). At the intersection, the non-adiabatic coupling terms (NACT) between the two potential energy surfaces become infinite and the Born-Oppenheimer approximation breaks down.[42]

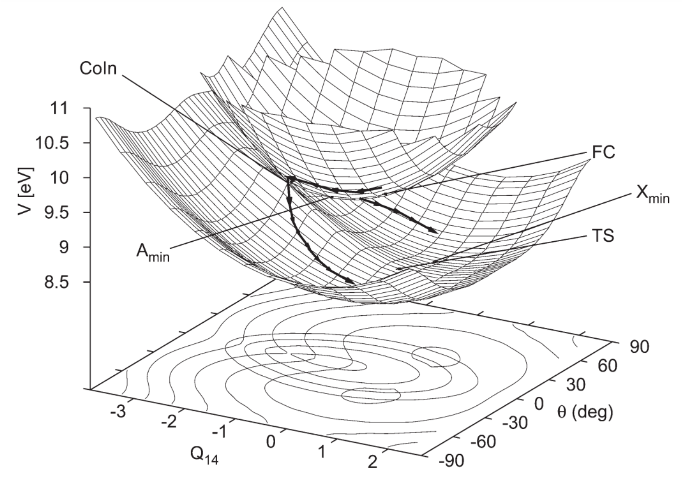


Figure 8: 3D surface plot of the conical intersection between two potential energy surfaces in the butatriene cation.[38]

As such, the vMCG method must be run in the diabatic picture, where the corresponding PES is smooth. Transformations between the adiabatic and diabatic surfaces are also performed on-the-fly using a diabatisation scheme, which relies on a transformation matrix, $\mathbf{K}$. If we define the NACTs between adiabatic states ψ_i with energy V_{ii} and ψ_j with energy V_{jj} as

$$F_{\alpha,ij} = \frac{\langle\psi_i| \nabla_\alpha \hat{H} |\psi_j\rangle}{V_{jj} - V_{ii}} \tag{56}$$

where ∇_α is with respect to the coordinate R_α, then we can use the approximation

$$\nabla\mathbf{K} \approx -\mathbf{F} \cdot \mathbf{K} \tag{57}$$

with $\mathbf{F}$ the matrix of NACT vectors. The equation is only exact if the basis set of electronic states is complete (infinite), which is not practical. Integrating (57) between two geometries at $\boldsymbol{x}$ and $\boldsymbol{x} + \Delta\boldsymbol{x}$ gives the transformation matrix at the next step

$$\mathbf{K}(\boldsymbol{x} + \Delta\boldsymbol{x}) = \exp\left(-\int_{\boldsymbol{x}}^{\boldsymbol{x}+\Delta\boldsymbol{x}} \mathbf{F}(\boldsymbol{x}') \cdot \mathrm{d}\boldsymbol{x}'\right) \mathbf{K}(\boldsymbol{x}), \tag{58}$$

which concludes the scheme.[43] In the *Quantics* package, the QC database is implemented using *SQLite*, where the tables for the diabatic gradient, Hessian, and PES are named `gra`, `hes`, and `pes` respectively, adiabatic data is stored in the `agra`, `ahes`, and `apes` tables, the NACT vectors in `nact`, and the adiabatic-to-diabatic transformation in `trans`.

3 ***Design***

3.1 Application programming interface

In order to start to create a GUI we first need to find a way to connect to existing analysis programs written in Fortran. Figure 9 shows the analysis process currently implemented. The middle route is the method we used in §1.2 to plot the absorption spectrum with `autospec`: a Fortran program reads the output of a *Quantics* simulation and outputs another file which can be read and plotted manually in *gnuplot*.

Alternatively, if the Fortran program implements a menu, the program can directly call *gnuplot* for plotting, without outputting any files. However, these menu programs offer an option to output the data used for plotting in a file for manual inspection of the data, which is particularly useful for our analysis GUI, as we generally do not want the analysis program to open a *gnuplot* window when we call it.

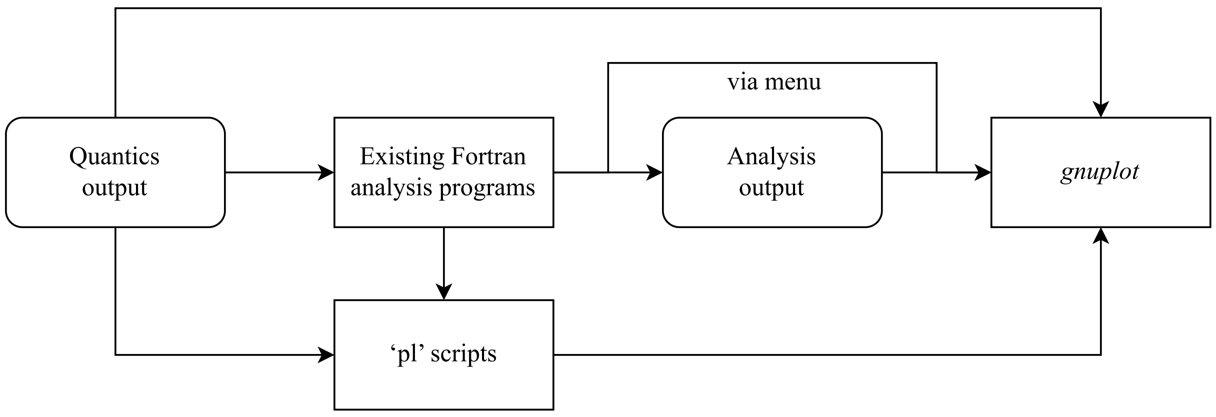


Figure 9: Flow chart of dependencies for the currently implemented analysis system in *Quantics*. Rounded boxes indicate files, and square boxes indicate programs.

The 'pl'-scripts offer a different route. Depending on its purpose, the script calls a Fortran program and reads its file output, or reads the *Quantics* simulation output directly, before calling *gnuplot*. Finally, the top arrow indicates *gnuplot* can also plot some of the simulation output directly.

The relationship between the analysis program and *gnuplot* can be called an interface, specifically, an application programming interface (API).[44] *Quantics* does not implement *gnuplot*—it only uses it to visualise the file outputs of the analysis (or temporary file for 'pl'-scripts and menu programs). Such is the basis of the analysis GUI. It can form an API to the existing Fortran programs through its file output. Thus, we can re-route Figure 9 as follows.

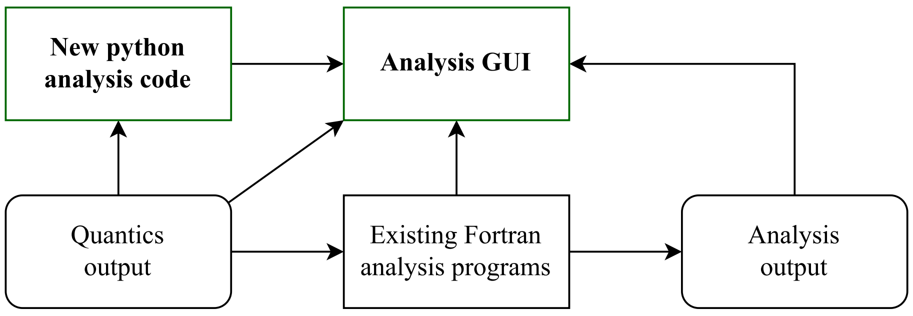


Figure 10: Ideal analysis system for *Quantics* using a GUI.

The analysis GUI can interface with the existing Fortran programs by calling it, similarly to the 'pl'-scripts, then read and plot the output directly within the GUI. If the original *Quantics* simulation output can be read directly, this can be implemented directly without an analysis routine being called. Lastly, as mentioned in §1.3, the creation of new analysis code in Python can also interpret the *Quantics* output and also can be fed into the GUI without using an intermediate file.

3.2 User interface design

The key to good user interface (UI) design is *usability*.[45] It is the extent a user can achieve their goals within the user interface based on three principles:

1. *Effectiveness*, how well a user can achieve their goals within the UI without running into errors, being lost, or needing help;
2. *Efficiency*, how much effort needs to be made in order to achieve the specified goals;
3. *Satisfaction*, how comfortable the user feels when interacting with the UI.[46]

In general, these are only factors to keep in mind when designing the UI, but can be quantified using a survey or questionnaire, which is talked about later in §5.3. Before thse principles can be satisfied, however, we must understand what the goals mentioned in these principles are. For the purposes of the analysis GUI there are three tasks the user should be able to perform:

- The user should be able to choose a directory, selecting a folder with the *Quantics* output files;
- The user should see a list of analyses, choose an analysis and enter the parameters of the analysis;
- The user should visualise the analysis, either in text or in a graph/plot.

To build a GUI around these tasks, it would be helpful to make a *content diagram*, shown in Figure 11. The second, third, and bottommost lines correspond to the three tasks just mentioned. Since an analysis output can be text-based or plot-based, we have a tab that can switch between the two. While it may be simpler to have one output view for both text and plot based analyses, the Fortran programs for plot-based analysis may also write a `.log` file or send additional information to the command line. This information can be written to the text widget for the user to view.

A secondary aim of the content diagram is to help in conceptual design, and allows the designer to see

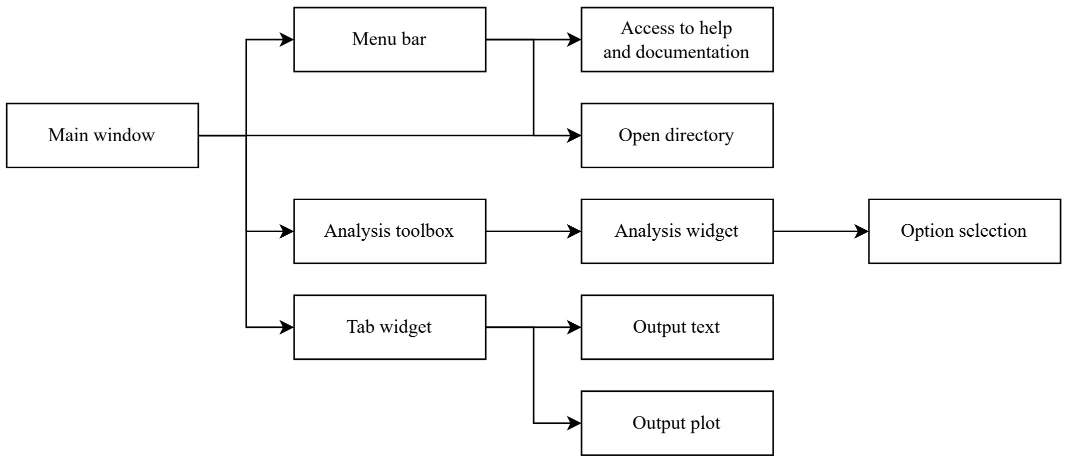


Figure 11: A conceptional content diagram of the GUI.

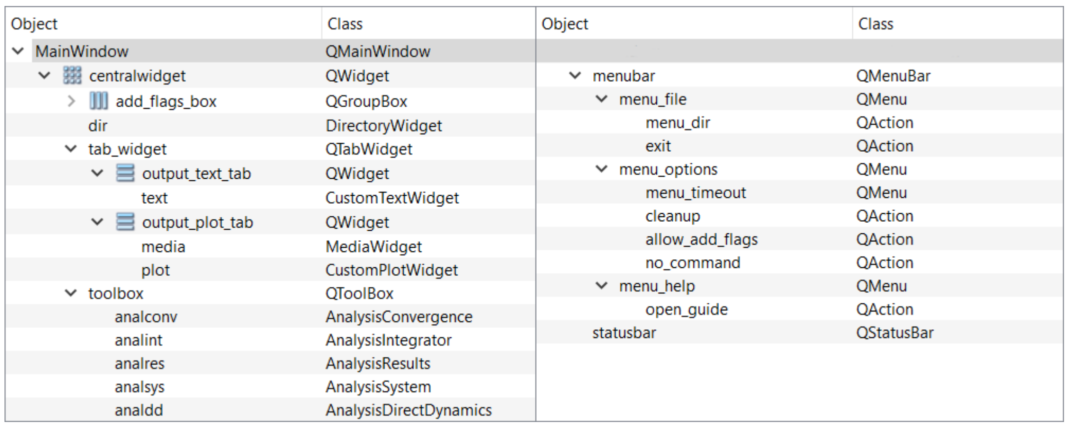
how usability can be maximised before working on the technical details. A number of heuristics exist for this purpose, though not all depend on solely the conceptual design. Some general principles that were kept in mind in creating Fig. 11 are:[47]

- **Consistency.** The directory can be written directly on the screen, which is the case for the `analysis` program, but also in the menu bar, which is consistent with many other GUIs.
- **Recognition, not recall.** An “option selection” box is included in the analysis widget which allows the user to select parameters for the analysis on the screen, rather than having to recall them.
- **Help and documentation.** In the menu bar, there should be a link to help and documentation on technical or less used aspects of the GUI, should they need it. Its implementation is explained in §5. However, it is also important that the design should allow the majority of users to achieve their goals without the use of further documentation.

To build a GUI using *PyQt*, a user has two options. They can use code, or use *Qt Designer*, a WYSIWYG (what you see is what you get) editor that allows elements of a GUI to be dragged directly where a person wants them. The use of Qt Designer also helps seperate the coding of the visuals itself from the coding of its implementation. As such, using *Qt Designer* would be a better choice. In general, objects in Qt Designer fall in one of two categories:

- **Composite widgets**, or containers, which may have other widgets inside of them. They can have a layout attached to them, which orders the subwidgets inside of them in a particular way (Horizontal, vertical, or a grid layout).
- **Input widgets**, which allows the system to obtain feedback from the user; for example, a *text edit* can obtain a string from the user, a *spinbox* can obtain an integer or a float, a *checkbox* can obtain a boolean (true or false), and a *combo box* or can obtain a choice from a number of predefined items.

Putting these widgets together, we can build something similar to Fig. 11 in Qt Designer.



Object	Class
MainWindow	QMainWindow
centralwidget	QWidget
add_flags_box	QGroupBox
dir	DirectoryWidget
tab_widget	QTabWidget
output_text_tab	QWidget
text	CustomTextWidget
output_plot_tab	QWidget
media	MediaWidget
plot	CustomPlotWidget
toolbox	QToolBox
analconv	AnalysisConvergence
analint	AnalysisIntegrator
analres	AnalysisResults
analsys	AnalysisSystem
analdd	AnalysisDirectDynamics

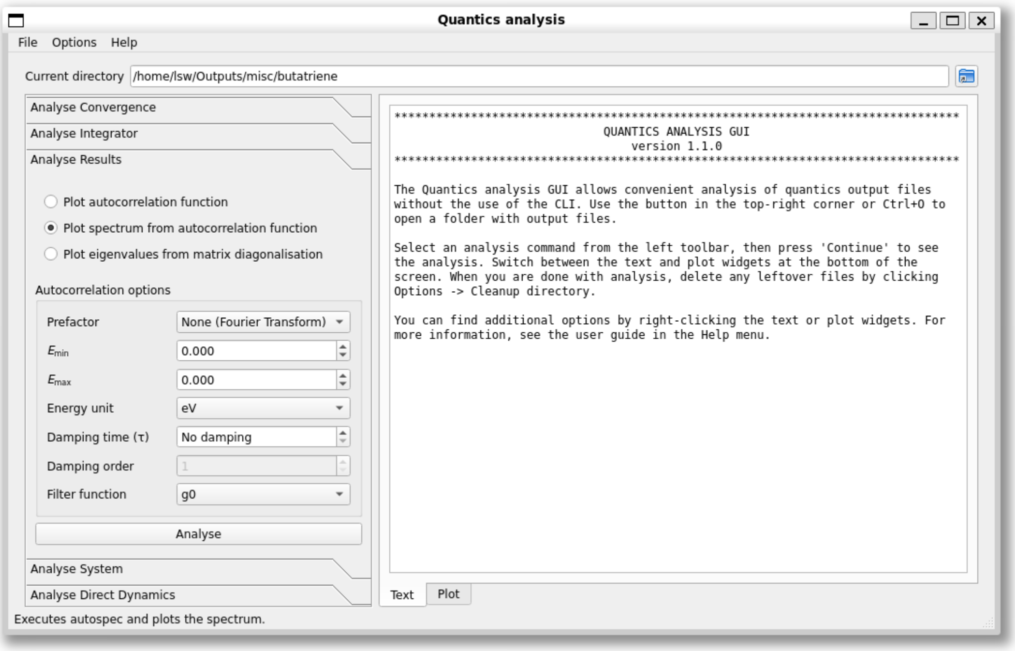
Object	Class
menubar	QMenuBar
menu_file	QMenu
menu_dir	QAction
exit	QAction
menu_options	QMenu
menu_timeout	QMenu
cleanup	QAction
allow_add_flags	QAction
no_command	QAction
menu_help	QMenu
open_guide	QAction
statusbar	QStatusBar

Figure 12: A tree view of contents as seen in the object inspector in Qt Designer.

The main window is made of a central widget with a grid layout as indicated by the icon to the left of its name. The analysis options are categorised into the four main sections per listing 3 (with the PES grouped with analyse system), as well as a fifth “Analyse Direct Dynamics” with new analyses explained in §4.12. The `MediaWidget` allows for control of animated plots (§4.9), and the contents of `add_flags_box` and `menu_options` are also additional features discussed in §4.13.

Some properties can also be set in Qt Designer, such as the margins or minimum size of a widget. A status tip property can be set on each analysis option, which displays a label on the status bar when hovering over it. This allows users familiar with the previous CLI-based analysis to know the name of the analysis program called by the GUI which they can consult the relevant part of the existing documentation.

The final layout of the GUI is shown in Fig. 13. The design is meant to be simple and uncluttered, so that the important aspects of the GUI stand out more. The task for visualising the analysis the most important



Current directory /home/lsw/Outputs/misc/butatriene

( ) Plot autocorrelation function
(●) Plot spectrum from autocorrelation function
( ) Plot eigenvalues from matrix diagonalisation

Autocorrelation options

Prefactor	None (Fourier Transform)
E_{min}	0.000
E_{max}	0.000
Energy unit	eV
Damping time (τ)	No damping
Damping order	1
Filter function	g0

```
********************************************************************************
                              QUANTICS ANALYSIS GUI
                                  version 1.1.0
********************************************************************************

The Quantics analysis GUI allows convenient analysis of quantics output files
without the use of the CLI. Use the button in the top-right corner or Ctrl+O to
open a folder with output files.

Select an analysis command from the left toolbar, then press 'Continue' to see
the analysis. Switch between the text and plot widgets at the bottom of the
screen. When you are done with analysis, delete any leftover files by clicking
Options -> Cleanup directory.

You can find additional options by right-clicking the text or plot widgets. For
more information, see the user guide in the Help menu.
```

Executes autospec and plots the spectrum.

Figure 13: The analysis GUI as shown on startup.

aspect of all the tasks, so it takes us the most space. At startup, no analysis can be visualised yet, so the GUI defaults to showing a concise description of what the GUI does.

Secondly, a list of analyses is shown of the left, seperated by the five categories mentioned above. Clicking each tab reveals a list of radio buttons, which correspond to the individual analyses. The user can input parameters for the analysis into the box below the radio button list. When the button 'Analyse' is pressed, the analysis is performed and visualised in the text or plot widgets.

Finally, a short line dedicated to showing the current directory is shown at the top, below the menu bar.

3.3 Object oriented programming

Now that the design of the UI is complete, we need to add functionality to the GUI. In Qt and PyQt, this is done using signals and slots.

- **Signals** are emitted when a widget on the screen is interacted with such that its internal state changes. When a signal is emitted depends on the widget and the interaction. For example, a button emits a signal when it is pressed, and a text edit emits a signal when its contents are modified, and a seperate signal if enter is pressed.
- **Slots** are the corresponding functions or methods that are executed when a signal is emitted. These functions are otherwise completely ordinary functions: nothing is special about them except that they are connected to a signal. The `@QtCore.pyqtSlot` decorator can be used to indicate that the method is a slot, but this is completely optional.

A signal and slots editor is present in Qt Designer but its functionality is severely limited: not all signals are available and slots can only take the shape of predefined functions. Thus, we have to use code to create slots and manually connect them to signals.

Objects in a GUI can naturally be represented by an object in a code, which are the underlying basis of object oriented programming (OOP). An object has attributes, or instance variables, which describe its internal state (e.g. the current value in a spinbox), and methods, which are functions that have access to instance variables (e.g. change the value in a spinbox).[48]

The definition of an object is given by a class, which can be thought of as a template or blueprint of an object. We can import the UI file created by Qt Designer into a class representing the main window, which is called `AnalysisMain`, using the following code.

```python
from pathlib import Path
from PyQt5 import QtWidgets, uic
class AnalysisMain(QtWidgets.QMainWindow):
    def __init__(self):
        # call the inherited class' __init__ method
        super().__init__()
        # load the .ui file into the current class
        uic.loadUi(Path(__file__).parent/'main_window.ui', self)
```

Listing 4: Code snippet for loading a UI file.

The use of the `Path` object allows the .ui file to be loaded from the Python file's directory. Otherwise, `uic.loadUi` tries to find it in the working directory that the user executes the Python file, which will likely not be the Python file's directory.

We can then place all the code/functionality of the GUI in this class. However, this is generally a bad idea because the resulting file will be very large in size, making organisation of the code and finding relevant methods or attributes harder, which in turn makes it harder to make modifications or add new features. Instead, we can split the code by responsibility, and have each class represent that responsibility. For example, the class `DirectoryWidget` has the single responsibility of storing and changing the directory that the GUI looks in to find output files. This is what is known as the *single responsibility principle*, a key design principle in OOP.[49]

These classes can then be interlinked using two well-known methods:

- **Inheritance** allows the attributes and methods in a class, the parent class, to be used in another class, the child class, via inheritance. The child class can then add new attributes and methods which may use the parent class' attributes and methods.
  This is known as an "is-a" relationship; for example, the `QDialog` class handles a pop-up in a GUI to communicate with the user, like asking a question or as a notification. The `QFileDialog` class, used in §4.1, inherits from `QDialog`. It is also a pop-up, but specialises in asking the user to select a file or directory. Thus, a `QFileDialog` "is a" `QDialog`.
- **Composition** allows a child object to be an attribute of a parent object. The child object is contained within the parent class, and attributes and methods can be accessed and executed from the parent class.
  This is known as a "has-a" relationship; for example, a `QWidget` is a container than can have multiple `QRadioButton`s within it. Thus, the `QWidget` "has a" `QRadioButton` (or multiple of them). When a UI is loaded into a class via listing 4, the attributes (child objects) are set automatically, and can be accessed via the object name in Qt Designer (Seen in Fig. 12).

These relationships can be visualised using a *class diagram*. The class diagram of the analysis GUI is shown in Fig. 14.

Each box represents a class. The first section has the name of the class, the second its attributes, and the third its methods. Inheritance is marked using an arrow with an triangular arrowhead from the child class to the parent class, and composition is marked using an arrow with a diamond arrowhead from the child object to the parent object. Section §4 details the implementation for each class.

There is one downside in not having all the code contained in one class, and it is that each promoted composite widget requires its own .ui file. Qt Designer only works with one .ui file at a time, and so it is not possible to see the contents of a composite promoted widget from the main window's .ui file, though multiple .ui files can be opened at the same time. This has the effect of making the UI design slightly harder to visualise.

3.4 Packaging

In general, each file should only contain the details for one class, as this allows it to be easier to find the class for modification. These files can also be grouped into different folders depending on their purpose. In particular, the classes that perform analyses are placed in their own directory called `analysis/`, tests are placed in `tests/`, while the rest are placed in a directory called `ui/`.

```
analysis_gui/
===============
ui/                ui/
analysis/          ===============    analysis/
tests/             __init__.py        ==================    tests/
__init__.py        main_window.py     __init__.py           ===========
gui.py             main_window.ui     convergence.py        fixtures/
environment.yml    dir_widget.py      convergence.ui        __init__.py
README.md          dir_widget.ui      integrator.py         run_tests.py
                   media_widget.py    integrator.ui         test_misc.py
                   media_widget.ui    results.py            test_analyses.py
                   analysis_tab.py    results.ui
                   custom_plot.py     system.py
                   custom_text.py     system.ui
                   coord_select.py    direct_dynamics.py
                                      direct_dynamics.ui
```

Listing 5: Directories and files in the `analysis_gui` folder.

In order to allow classes in the folders to gain access to the contents of the other folder, we must make a *Python package* by placing a file named `__init__.py` in each directory, as well as the directory above it. We can

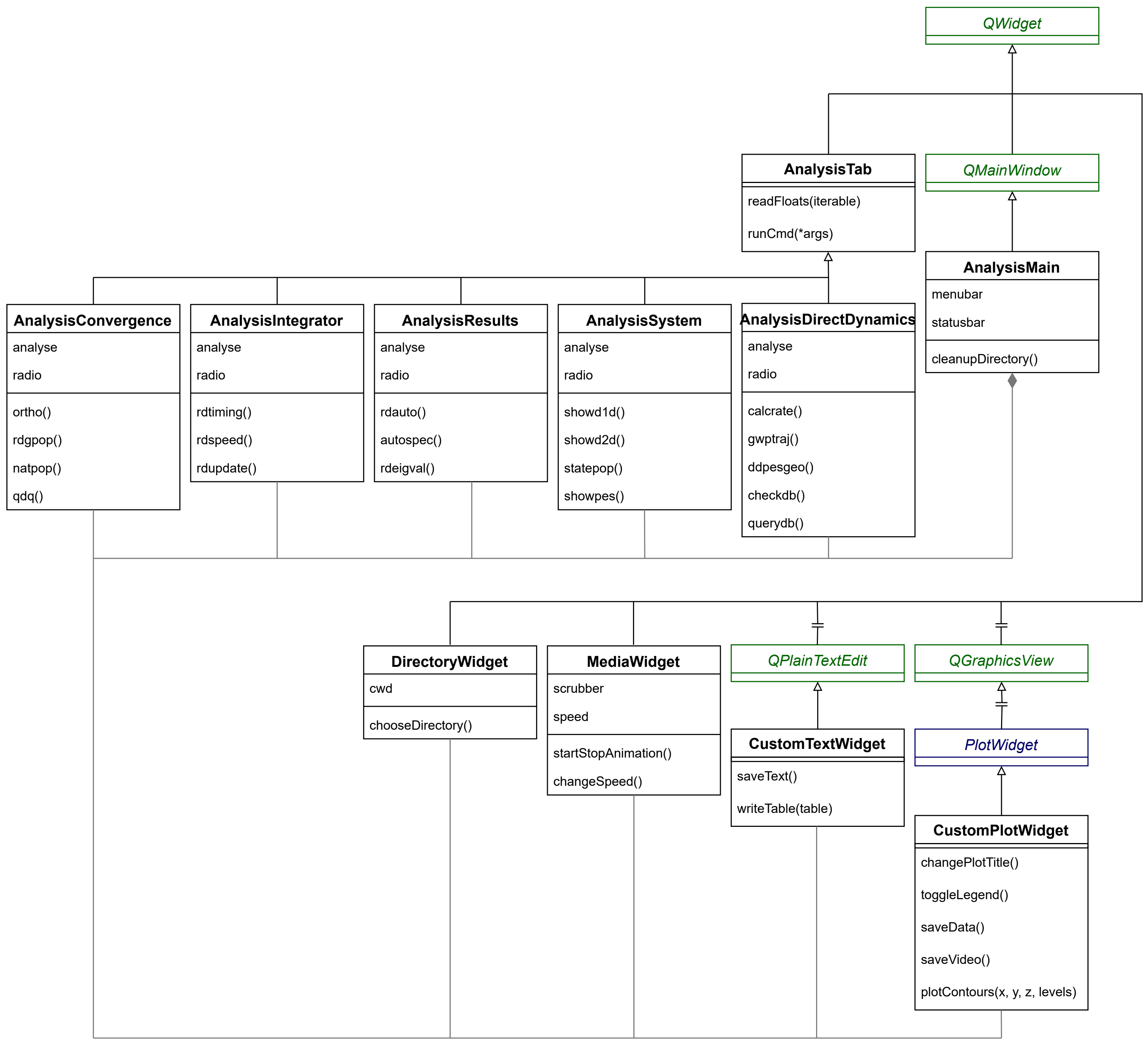


Figure 14: Class diagram (UML specification) of the analysis GUI. Not all attributes and methods are shown. Green classes are implemented by *PyQt*, the blue class by *pyqtgraph*. The "=" symbol means some layers of inheritance are hidden.

then use relative imports to obtain the classes in another folder. For example, in `analysis/convergence.py`, we can import the `AnalysisTab` class using `from ..ui.analysis_tab import AnalysisTab`, where the `..ui` looks for the folder `ui/` from one directory up.

Making a Python package also has a second purpose. In order to use inherited widgets in Qt Designer, you must "promote" the widget, by inserting the parent class into the GUI instead, then filling a form detailing the name of the child class and the location of the module (file) that the class can be imported from. For the latter, this can be made easier if all the files were located in a package. For example, using the directory structure in listing 5, the location of `convergence.py` can be specified as `analysis_gui.analysis.convergence`.

Manual editing of the .ui files are required to conform with the schema in Fig. 14. Firstly, promoted `QWidget`s are considered containers automatically by Qt Designer, which can be changed by manually removing `<container>1</container>` from the `<customwidgets>` section of the .ui file.

Secondly, it is not possible to promote the top-level widget in Qt Designer. In the .ui files in the `analysis/` directory, the top-level widget is required to be `AnalysisTab`, otherwise the `PyQt5.uic` module will complain that the class name is incorrect. To fix this, the `class` tag in the top-level widget can be changed manually in the .ui file, i.e. from `<widget class="QWidget">` to `<widget class="AnalysisTab">`, assuming the `AnalysisTab` class is detailed in the `<customwidgets>` section.

We can then create and open the GUI application as follows.

```python
def openGui():
    app = QtWidgets.QApplication(sys.argv)
    # create and open the window
    window = AnalysisMain()
    window.show()
    # run the main Qt loop
    sys.exit(app.exec_())
```

Listing 6: Code snippet of opening the main window using PyQt.

This code is included in `gui.py`, but executing the code directly from this file will not work. This is because the `gui.py` file is inside the package, and Python does not consider the directory a Python file is executed in as a package (Python's creator, Guido van Rossum, considers this to be an "anti-pattern", i.e. something to avoid[h]). Thus all relative imports will fail. Instead, we can specify another file outside of the package (namely `quantics_analysis_gui.py`) which imports the `openGui` function of `gui.py` and runs it.

Finally, we must ensure the `analysis_gui` package is located on `$PYTHONPATH`, such that the `PyQt5.uic` module can find the package. This can be done by appending the folder location to the `sys.path` list. Once this is complete, it is possible to open the GUI from any Python file in any location using `from analysis_gui import gui` and `gui.openGui()`.

3.5 Code style

The final design choice is the use of code style. In Python the use of the PEP8 style guide is encouraged.[50] However, the Python code makes large use of the PyQt package, which in general does not adhere to PEP8. If the code did use PEP8, this would make the code style inconsistent, which breaks the purpose of adhering to a style guide in the first place.

	PEP8	**PyQt/pyqtgraph**	**analysis_gui**
Attribute/Variable	`snake_case`	(*inconsistent*)	`snake_case`
Method/Function	`snake_case`	`camelCase`	`camelCase`[i]
Class	`PascalCase`	`PascalCase`	`PascalCase`
Module/File	`snake_case`	(*inconsistent*)	`snake_case`

Table 1: Code style recommendation for PEP8, Code styles used in PyQt/pyqtgraph, and code style chosen for `analysis_gui`.

Table 1 shows the format used in PEP8 and PyQt/pyqtgraph. The code for the analysis GUI uses the PyQt/pyqtgraph convention where possible, and PEP8 for all other cases.

4 Implementation

4.1 Changing the directory

The first task is to embed a directory navigation widget into the GUI, so that the user knows what directory the analysis GUI is looking for output files in. This is done using the directory widget, which has an input widget `QLineEdit` for this purpose. This also allows the directory to be typed in by the user. At startup, the line edit is set to the current working directory using `Path.cwd()`.

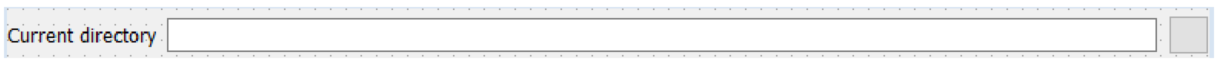


Figure 15: The directory widget as it appears in Qt Designer. The icon of the button cannot be set in Qt Designer so it appears blank; it is instead set in the code.

The goal of the widget is to provide the manipulation of the widgets' `cwd` attribute. This is a Path object that, for example, can be accessed by an analysis method to find an output file using `DirectoryWidget.cwd/'filename.txt'` (For Path objects, the forward slash appends to the path). Since the `QLineEdit` stores the directory, the `cwd` getter can simply return this text.

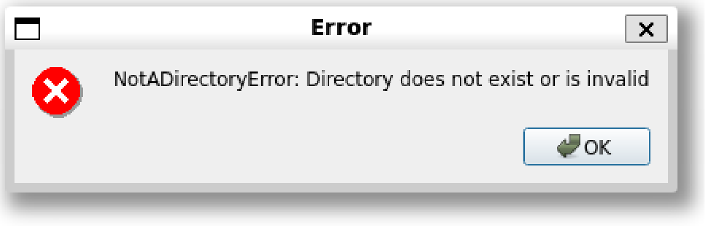
When the user enters a directory using the line edit, we can check that it is a valid directory by connecting the line edit's `editingFinished` signal to `cwd`'s setter that uses `Path.is_dir` method. If it is not a valid directory, then we can revert to the previous valid directory by using `QLineEdit.undo()`, then display an error

[h]`https://mail.python.org/pipermail/python-3000/2007-April/006793.html`

[i]Except methods in the analysis widgets, which are named after the programs they call, making it easier to understand their purpose.

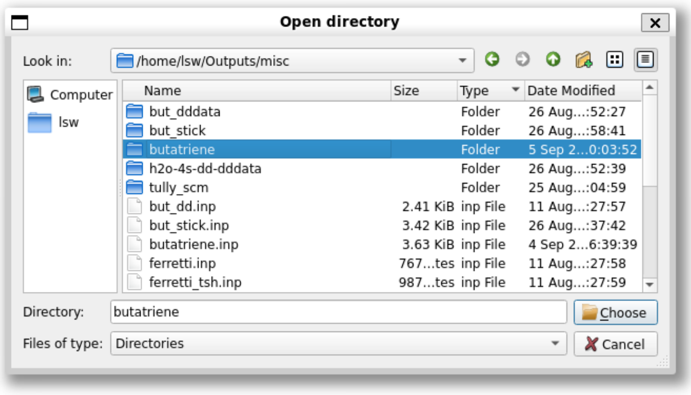
message using `QtWidgets.QMessageBox.critical`, which take the parent widget, title text, and message text as arguments (Fig. 16).

The second task is to allow the user to select a directory using a pop-up window, which can show the user's directory structure, activated by clicking on the button on the far right (or `Ctrl+O`/using the menu bar). PyQt has a widget for this purpose, known as `QtWidgets.QFileDialog`. To select a directory, we can use the `getExistingDirectory` method, which creates a new window per Fig. 17. By default, this only shows the user's directories; it would be useful if it can show files as well, which can be done by passing it an empty options object `options=QtWidgets.QFileDialog.Options()` which overrides the default option `QtWidgets.QFileDialog.ShowDirsOnly`. Once the directory is selected, the new directory is passed to `cwd`'s setter.


Error

NotADirectoryError: Directory does not exist or is invalid

Figure 16: Error popup from inputting an invalid directory.


Open directory

Look in: /home/lsw/Outputs/misc

Name	Size	Type	Date Modified
but_dddata		Folder	26 Aug...:52:27
but_stick		Folder	26 Aug...:58:41
butatriene		Folder	5 Sep 2...0:03:52
h2o-4s-dd-dddata		Folder	26 Aug...:52:39
tully_scm		Folder	25 Aug...:04:59
but_dd.inp	2.41 KiB	inp File	11 Aug...:27:57
but_stick.inp	3.42 KiB	inp File	26 Aug...:37:42
butatriene.inp	3.63 KiB	inp File	4 Sep 2...6:39:39
ferretti.inp	767...tes	inp File	11 Aug...:27:58
ferretti_tsh.inp	987...tes	inp File	11 Aug...:27:59

Directory: butatriene

Files of type: Directories

Figure 17: Dialog to select a directory.

4.2 The `AnalysisTab` widget

The purpose of the `AnalysisTab` widget is to provide a framework that allows analysis methods to be implemented or created more easily. Each analysis tab (Fig. 13) inherits from this class (Fig. 14) in order to obtain useful methods and have its basic functionality automatically implemented. Since each analysis tab shares many common features, this inheritance allows for code reuse.

In particular, each analysis tab shares common widgets: a list of radio buttons in a `QWidget` which represent a choice of which analysis to perform, and a push button `QPushButton` which performs the selected analysis. These widgets are created in Qt Designer and loaded into the inherited class in the class' constructor (the `__init__` method)

If we standardise the names of these widgets as `self.radio` and `self.analyse`, respectively, then basic signal-slot selection can be performed in the `AnalysisTab` class instead of manually for each inherited class. For example, if a mapping (dictionary) between the selected radio button's index and the method to execute when `analyse` is pressed is given in the inherited class, then the functionality of performing an analysis can be done using listing 7.

```python
# in constructor, mapping defined between radio button index and method objects,
# e.g. self.methods = {0: self.analysisMethod0, 1: self.analysisMethod1}

# get index of checked radio button (there should only be 1)
radio_index = [index for index, radio in enumerate(self.radio) if radio.isChecked()][0]
# call method associated with index
self.methods[radio_index]()
```

Listing 7: Code snippet of the `analysePushed` method.

However, a problem occurs if the button is pushed while an analysis is being performed. For analyses that take up more than a few seconds, clicking the button again will queue another method call in the system, i.e. if the button is clicked twice, a redundant second method call will be performed. Ideally, the user should know the system is busy performing the analysis and the button should be frozen to prevent any further presses. This can be implemented using listing 8. Note, the `repaint()` method needs to be called otherwise the button's status update is delayed until the next event loop, meaning that nothing happens—calling `repaint()` updates the button immediately.

If the analysis method called raises an exception, the `analysePushed` method terminates early and the button will not unfreeze, preventing the user from doing anything. Therefore, the analysis method call is also wrapped in an try-except block, with the nature of the exception given to the user via `QtWidgets.QMessageBox.critical`. The complete traceback of the exception can then be given to the developer for debugging using `print(traceback.format_exc(), file=sys.stderr)`.

```
QtWidgets.QApplication.setOverrideCursor(
    QtCore.Qt.WaitCursor # set cursor to 'wait'
)
self.analyse.setEnabled(False)
self.analyse.setText('Busy')
self.analyse.repaint()
# ... method call ...
QtWidgets.QApplication.restoreOverrideCursor()
self.analyse.setEnabled(True)
self.analyse.setText('Analyse')
```

Listing 8: Code snippet of freezing the push button until analysis is complete.

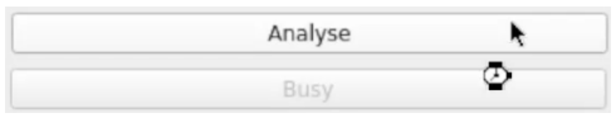


Figure 18: Top: The system is available to perform an analysis. Bottom: The system is busy performing an analysis. The cursor icons for both situations are also shown.

4.2.1 Checking a file exists

In addition to performing an action when `self.analyse` is pressed, we can also perform actions when a radio button in `self.radio` is pressed. This is useful since this allows us to change the state of the UI when a specific analysis option is selected. For example, most analyses require a certain output file to be present in the directory—if that file is not present, then we can disable `self.analyse` from being pressed.

Like the methods dictionary, these files can be given in the inherited class via a mapping, with a list of filenames in the current directory as the value. Firstly, we connect all the radio buttons to a slot method named `optionSelected` (Listing 9), which for now only calls `checkFileExists`, but will have another purpose later (§4.6). Then, in the `checkFileExists` method, a list of missing filenames can be generated using a list comprehension `[file.name for file in filenames if not file.is_file()]`. If this list has any members, they are displayed on the frozen "Analyse" button (Fig. 19).

```
for index, radio in enumerate(self.radio):
    if radio.isChecked():
        self.checkFileExists(index)
```

Listing 9: Code snippet of the file checking functionality of the `optionSelected` method.

Figure 19: The "Analyse" button frozen, showing the missing files that are required for the chosen analysis.

4.2.2 Reading values from a file

The core method of interfacing with the Fortran analysis program is through the analysis output files. These files are often in the form of a grid of values (Listing 10).

```
0.00     3.0171885631E+02
1.00     2.8208406002E+01
2.00     3.0443131784E-03
3.00    -1.9018896319E+00
4.00    -1.6790777029E-01
```

Listing 10: A grid of numbers produced from an analysis program. As an example, the first column might represent time, and the second might represent some other quantity.

It would be useful to implement a function which can read these files in `AnalysisTab` and produce a matrix of floating point numbers (a computer's representation of a real number) from it. A suitable data type for this would be a *NumPy array*. *NumPy* is a Python module that provides support for multi-dimensional arrays and mathematical functions to operate on these arrays.[51] The inherited classes' analysis methods can then plot this array, or manipulate it using NumPy's extensive library for scientific computing.

To create the NumPy array, it needs to be passed a list of lists: one representing rows, and one representing columns (or n nested lists for an n-dimensional array). Each row is located on a new line, so we can simply use a for loop to iterate over each line. Then for the cells in the rows, we can use *regular expressions* (regex), which use a "pattern" to find sequences of characters which can then be turned into floats.

Initially, the code used a regular expression that only matches valid floating point number forms. The text would be converted to a float if and only if the text matches the float regex, such that we know the text-to-float parsing will succeed. This is known as a *look before you leap* (LBYL) approach.

There is another approach in that we could use a regular expression that matched any sequence of characters (except whitespace). We do not know if text-to-float parsing will succeed, but if it does not, we can ignore that line. This is known as a *easier to ask for forgiveness than permission* (EAFP), characterised by a try-except block over LBYL's `if` statement.[52] The code for both approaches is given in listing 11. Explanation for the regexes are given in §8.1.

```python
def lbyl(iterable) -> np.ndarray:
    data = []
    regex = r'([+-]?\d+(?:\.\d*)?(?:[eE][+-]?\d+)?|\.\d+)'
    for line in iterable:
        matches = re.findall(regex, line)
        if matches:
            data.append(list(map(float, matches)))
    return np.array(data)

def eafp(iterable) -> np.ndarray:
    data = []
    regex = r'\S+'
    for line in iterable:
        matches = re.findall(regex, line)
        try:
            data.append(list(map(float, matches)))
        except ValueError:
            pass
    return np.array(data)
```

Listing 11: LBYL and EAFP implementations of reading a grid of numbers. `iterable` can be a file object or a list of strings.

Due to the relative complexity of the LBYL's float regex, it is assumed that the EAFP approach would be faster. To confirm this, the performance of both implementations were tested against a grid of 4000×4 floats, stored in a file. Each function parsed the file 1000 times. The time for the LBYL implementation was 21.69 s, while the EAFP implementation took only 14.59 s. As such, the EAFP approach was used in the code.

The `readFloats` method has a couple of additional optional parameters for flexibility. Firstly, if we expect that a file has a set number of columns, we can specify this in the `floats_per_line` parameter, such that if the format of the file has been updated, the code will fail on purpose, serving as an indicator to the developer to update the outdated code. This is also useful to prevent accidental errors with inhomogenous column lengths being parsed (e.g. one row having two columns, the other having three).

The second parameter is provide an optional "ignore regex". If a line matched this regex, the line would be ignored. This may be useful in a situation where the output files is in a *gnuplot*-format where a # symbol indicates a comment which should not be read. If the regex `r'^#'` is passed to this parameter, lines starting with # will be ignored.

4.3 The plot widget

To visualise analyses, a plot widget is implemented inside the GUI. This allows for a more native experience compared to something like *gnuplot* which opens in a new window. In Python, the most popular package for plotting and visualisations is *matplotlib*. However, *pyqtgraph* was chosen to be the tool used to show plots and graphs since it integrates much more nicely with PyQt, and also has the advantages of being faster, more interactive, and providing a more understandable API leading to simpler code.

As an example of the last point, *pyqtgraph* provides an interactive plot widget simply by adding `pyqtgraph.PlotWidget` as a component of the main window, while in *matplotlib*, you must import `matplotlib.backends.backend_qt5agg.FigureCanvasQTAgg` for the plot canvas, then `matplotlib.backends.backend_qt5agg.NavigationToolbar2QT` to provide interactivity. However, *matplotlib* is a much more comprehensive package with a lot more features; *pyqtgraph* lacks certain abilities like contour plots which have to be implemented manually (see §4.11).

The interactions the user can make with the plot widget implemented by default in *pyqtgraph* are:

- Ability to move the data on the screen by left-click and drag as well as zoom in on the data by right-click and drag or scroll wheel. User can also select "one-button" mode on the menu which allows the user to drag a box around the part of the graph they want to zoom into (Fig. 20).
- Ability to show or hide each line in a multiline plot by clicking on the respective entry in the legend.
- Ability to perform built-in transformations to the data, such as $\mathrm{d}y/\mathrm{d}x$, and conversion from linear plot to a log plot.
- Ability to export still images of the current plot as well as data in the plot as a .csv file (Fig. 21).

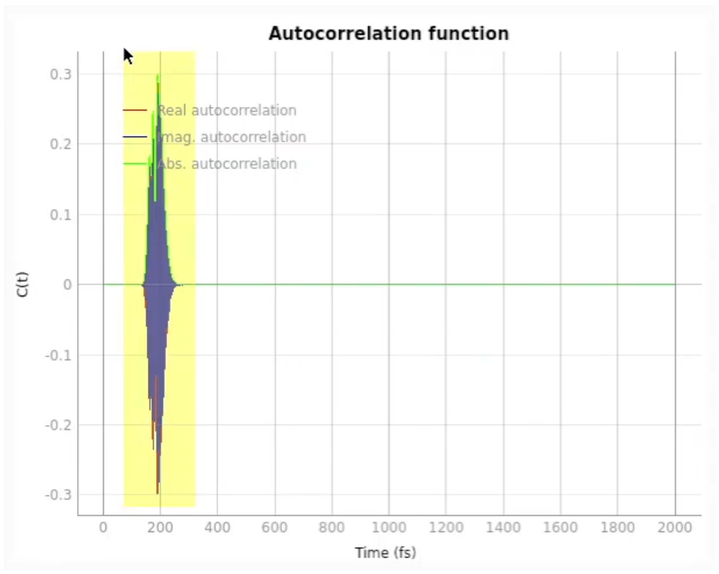


Figure 20: Dragging an area to zoom in on an interesting part of the plot in "one-button" mode.

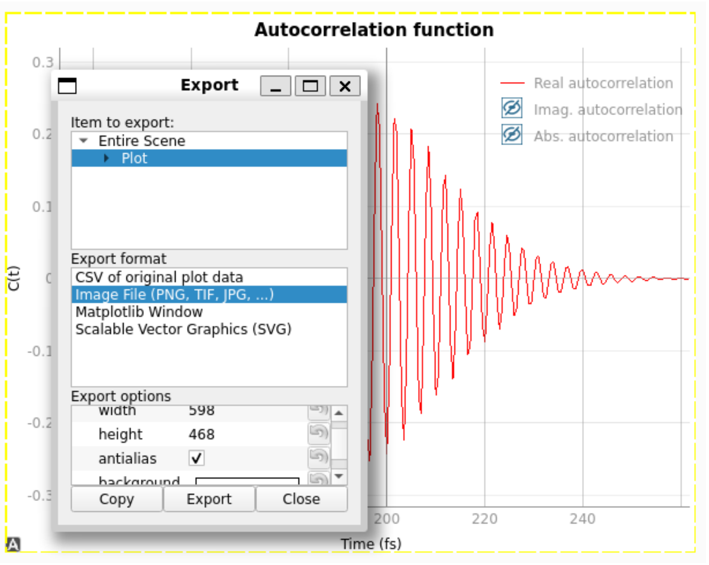


Figure 21: Saving an image using *pyqtgraph*'s built-in exporting option.

Analysis methods can call methods on the plot widget to create visualisations. However, the plot widget is a component of the main window, not a component of the analysis tab, so they must have a pointer to the main window to access the plot widget. This is given by PyQt using the `window()` method, such that the plot widget is accessed at `self.window().plot`, and the directory widget is accessed at `self.window().dir`.

To show a line plot using the plot widget, we simply use the `plot` method per `self.window().plot.plot(x_data, y_data, name=name, pen=pen)` where `x_data` and `y_data` are arrays, `name` is the corresponding label in the legend, and `pen` sets the style of the line, e.g. `pen='r'` makes the line red.

If the user chooses to perform multiple analyses, the plot has to be refreshed to prepare for replotting. By inheriting from pyqtgraph's `PlotWidget`, we can add additional methods that may be helpful in the implementation of analysis methods, such as `reset()`, which clears the current plot and resets axis labels. This method will have additional purposes discussed later on.

We have now covered all the basic tools to create a basic analysis method. Assuming that we have created a .ui file, created a class for this .ui file that inherits from `AnalysisTab`, then adding the method name to `self.methods` and the required files to `self.required_files`, we can write something in the form of listing 12:

```python
def genericanalysis(self):
    filepath = self.window().dir.cwd/'filename'
    with open(filepath, mode='r', encoding='utf-8') as f:
        self.window().data = self.readFloats(f, floats_per_line=n_col)
    self.window().plot.reset(switch_to_plot=True)
    self.window().plot.setLabels(title='title', bottom='xlabel', left='ylabel')
    # repeat for each column...
    self.window().plot.plot(self.window().data[:, 0], self.window().data[:, 1],
                            name='name', pen='r')
```

Listing 12: Code snippet of a generic analysis method that reads values from a file and plots them.

This method looks for a file named `'filename'` in the user-selected directory and reads a grid of values with `n_col` columns. We store the NumPy array returned from the `readFloats` method into an attribute in the main window because this allows us to more easily implement a method that can save this array to a user-defined location for future use (see §4.4).

To plot the this array, we first clear any existing graph using `reset()`. Since there are two output tabs (one for text, one for the plot), we can automatically switch to the text tab using the `switch_to_plot` parameter, which simply calls `self.window().tab_widget.setCurrentIndex(1)` (The plot tab is at index 1). To plot the second column against the first column, we can use *array slicing* in the form `self.window().data[rows_to_select, columns_to_select]` where a colon in `rows_to_select` selects all the rows.

The following subsections details specific variants of this basic code.

4.3.1 `rdauto`

If the keyword `auto` is given in the *Quantics* input file, *Quantics* will produce an `auto` file as an output. This file contains the *autocorrelation function* $C(t)$ defined as

$$C(t) = \langle \Psi(0) | \Psi(t) \rangle \tag{59}$$

which can be used to obtain the spectrum of a system (§4.6.4).

For symmetric Hamiltonians $\hat{H} = \hat{H}^{\mathsf{T}}$ and real initial wavepackets $\Psi(0)$ we can use

$$C(t) = \langle \Psi^*(t/2) | \Psi(t/2) \rangle \tag{60}$$

which is generally more accurate and only requires propagation for half the time for which the autocorrelation function is necessary.[53]

The format of the `auto` file is a grid of values with four columns, where the first column is time (in fs), and the second, third, and fourth columns are the real, imaginary, absolute values of the autocorrelation function at that time. When writing the code for this method, we can mention this a docstring, which is a description of what the code does. Listing 13 gives the docstring for the `rdauto` method.

```
'''
Reads the auto file, which is expected to be in the format, where each cell is a float,

t.1    re.1    im.1    abs.1
t.2    re.2    im.2    abs.2
...    ...     ...     ...
t.m    re.m    im.m    abs.m

where t is time, and re, im, abs are the real, imaginary, and absolute value of the
autocorrelation function. Headers are ignored. Plots the autocorrelation function.
'''
```

Listing 13: The docstring of the `rdauto` method.

Writing the docstring in this form allows future developers to know what format the file is expected to be in, allowing for easier debugging if the format of the file changes. Similar docstrings are present for all analysis methods covered in this report, but for conciseness only this one will be explicitly stated.

We can then use listing 12 to implement the rest of this method, where the filename is `auto` and `n_col` is 4. We can also set `ignore_regex='^#'` to ignore the header in the `auto` file. After setting the title and labels, we call `self.window().plot.plot` three times, plotting column 2, 3, 4 against column 1 for the real, imaginary, and absolute values of the autocorrelation function, respectively.

It may also be useful to the user if we write the contents of the auto file into the text widget for manual inspection. To do this we simply write `self.window().text.setPlainText(f.read())`, where `f` is the file object, followed by `f.seek(0)` such that `readFloats` can read the file from the beginning. Alternatively, for smaller files, we can set `txt = f.read()` and pass `txt.split('\n')` into `readFloats` instead.

4.3.2 `rdspeed`

The `speed` file is created after a *Quantics* simulation which reports the cumulative CPU and real time taken between each time step in the simulation. It has 6 columns: the propagation time (in fs), the CPU time (in seconds), the delta in CPU time (non-cumulative time between each time step), then the real time in a date format (`MMM DD hh:mm:ss`), in seconds, and in hours.

This format is slightly problematic for the `readFloats` methods due to the date column. Since the date contains text (specifically, for the month), parsing the file directly will fail since the method skips any lines that cannot be parsed into floats. The solution to this is simply to remove the date column using regex. We can match any date in the format `MMM DD hh:mm:ss` using `r'\w{3}\s+\d{1,2}\s+\d{2}:\d{2}:\d{2}'` (See §8.1 for an explanation) and replace them with blanks using `re.sub`. After this, the file can be parsed normally similarly to listing 12.

Only the CPU time and real time columns are plotted in this case, since the delta in CPU time can be retrieved using the in-built dy/dx transformation in the plot widget, and there is little point in plotting multiple lines for different units.

4.3.3 `rdupdate`

For *Quantics* simulations using a constant mean field (CMF) integrator (§2.4) with an adaptive update interval, the `update` file is created. The basic idea behind CMF is that the matrix element $\langle \Phi_J | \hat{H} | \Phi_L \rangle$ and the inverse density matrix and mean field product $(\boldsymbol{\rho}^{(k)})^{-1} \langle \mathbf{H} \rangle^{(k)}$ change much slower against time compared to the SPFs and the A-vector. Thus, they can be held constant for a longer propagation time, thus reducing computational cost, but not too long such that the error of SPFs and the A-vector passes a certain threshold.[24]

The update file contains five columns: the update step number, the update step size (time that the matrix element, etc. are held constant), the error of A, the error of the SPFs, and the propagation time (cumulative step size). However, not every row has the same number of cells. If the error of either the coefficients or the SPFs passes a certain tolerance, the step is repeated with a smaller step size. The same happens if the integrator fails to converge. In these cases, only the step number is included in that row (possibly with the error as well).

The `floats_per_line` parameter in `readFloats` is useful for these failed steps. If we set this parameter to 5, then any failed steps are automatically ignored. We can then plot the update step size or the errors against the propagation time. However, since these represent different quantities we do not want to show all of them in one plot. We can let the user choose which quantity to plot using a combo box. This essentially acts as a parameter of the analysis method, and is further discussed in §4.6.

4.3.4 `rdeigval`

Quantics also contains methods for solving the time-independent Schrödinger equation, namely via the diagonalisation of the Hamiltonian, though it is not based on the MCTDH methods and only works for smaller systems. If the keyword `diagonalise` is given in the input file, the `eigval` file will be created containing the eigenvalues from the diagonalisation (column 2, in eV), as well as intensities (column 3), error estimates of the eigenvalues (column 4), and the excitations (column 6, in cm^{-1}). The first column contains the number, and the fifth the eigenvalues in cm^{-1}, both of which are ignored. Like in §4.3.3, these are different quantities, and so the user chooses which column to plot using a combo box.

4.4 Customising the plot and text widgets

In the interactive CLI menus, options were available to the user to customise the aesthetics of the plot, such as toggling the visibility of the legend or the title (listing 2). Normally, this would not be possible by only using pyqtgraph's default `PlotWidget`, but we can inherit from this class to implement additional functionalities. We have already done this to implement a `reset` method in listing 12.

A suitable place for the user to change these settings are in the context menu, which can be made visible by right clicking on the widget. pyqtgraph has already some features in this menu such as transformations, manual setting of the x and y axes, grid visibility settings, etc.

To add options to these menus, we must acquire the underlying `QMenu` widgets. For the `PlotWidget`, the context menu is available at `PlotWidget.getPlotItem().vb.menu` and the "Plot options" submenu is available at `PlotWidget.getPlotItem().ctrlMenu` (See Fig. 22). To add an option, we can use something like `self.action = menu.addAction('Option name')`. This returns a `QAction` object which has a `triggered` signal that can connect to a slot. This code can be placed in the constructor of the inherited `PlotWidget`, named `CustomPlotWidget`.

The most basic example of this is to add an option to show or hide the legend, shown in listing 14, which connects to a slot `toggleLegend` that, depending on whether `self.legend_checkbox.isChecked() is True`, shows or hides the legend using `show()` or `hide()`, respectively.

```python
plot_menu = self.getPlotItem().ctrlMenu
self.legend_checkbox = plot_menu.addAction('Show Legend')
self.legend_checkbox.setCheckable(True)
self.legend_checkbox.setChecked(True)
self.legend_checkbox.triggered.connect(self.toggleLegend)
```

Listing 14: Adding a "Show legend" option into the context menu.

The second customisation is to allow the user to change the title to one that they can specify. The user can then add additional information that the default title set in the analysis methods does not detail. To place a `QLineEdit` inside a menu, we create a `QWidgetAction` object and input the `QLineEdit` as a parameter to `QWidgetAction.setDefaultWidget`. The `QLineEdit.textChanged` signal can be connected to a slot that changes the plot title using `CustomPlotWidget.setTitle`. In order to add a label to the `QLineEdit` we can place it inside a submenu, which can simply be done using `QMenu`'s `addMenu` method, which returns another `QMenu` that can be used as shown above.

Note that using the method described above means that if the title is changed by the user, the original title set by the analysis method cannot be recovered. To fix this, we add an attribute named `default_title` to `CustomPlotWidget` which can be changed by overriding (re-defining a method that already exists in the parent class) the `setLabels` method with the `title` parameter to change the `default_title` attribute instead. The slot that changes the title can then check if the `QLineEdit` is empty, in which case the slot changes the title to `default_title`—otherwise, the user-chosen title. If the user wants to clear the title completely, they can enter a empty space.

Though the intent of the GUI is to visualise analyses, it is not meant to create publication quality graphics. For users who need more customisation, touching up or modifying the results, we can also implement the ability to save the underlying NumPy array that the plot widget uses using `numpy.save`, which has the file's location and the NumPy array as parameters. While the in-built "Export" option can export an .csv of the currently plotted data, a NumPy array may be more convenient for Python users, and for animated plots (§4.9), the NumPy array will contain data for all frames rather than for the current frame only.

In the analysis method (listing 12) the NumPy array is stored at `self.window().data`. The user can then choose a file location using `QtWidgets.QFileDialog.getSaveFileName`, which brings up a window similar to Fig. 17. If the user selects an existing file as a save location, NumPy will actually append the data to this file; to overwrite instead, we simply delete the file first. After the data is saved, the user is given a confirmation popup using `QtWidgets.QMessageBox.information` that communicates to the user that the data has been saved successfully.

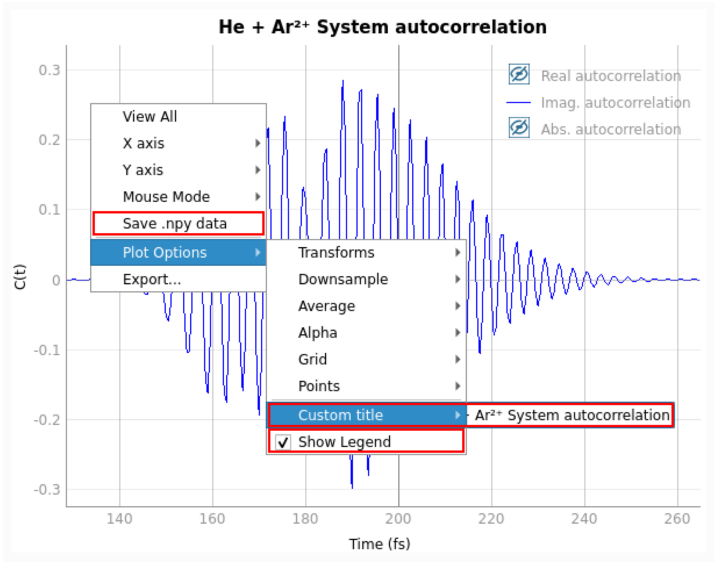
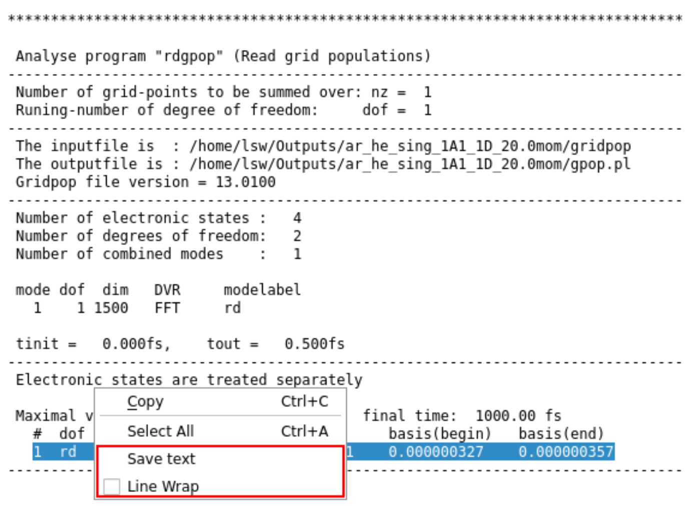


Figure 22: The context menu that appears when right-clicking on the plot widget. Additional options implemented highlighted in red. Not shown is the "Save video" option only visible for animated plots (§4.9).


```
******************************************************************************

 Analyse program "rdgpop" (Read grid populations)
------------------------------------------------------------------------------
 Number of grid-points to be summed over: nz =  1
 Runing-number of degree of freedom:     dof =  1
------------------------------------------------------------------------------
 The inputfile is  : /home/lsw/Outputs/ar_he_sing_1A1_1D_20.0mom/gridpop
 The outputfile is : /home/lsw/Outputs/ar_he_sing_1A1_1D_20.0mom/gpop.pl
 Gridpop file version = 13.0100
------------------------------------------------------------------------------
 Number of electronic states :   4
 Number of degrees of freedom:   2
 Number of combined modes    :   1

 mode dof  dim   DVR     modelabel
   1    1 1500   FFT     rd

 tinit =   0.000fs,    tout =   0.500fs
------------------------------------------------------------------------------
 Electronic states are treated separately

 Maximal v                                 final time:  1000.00 fs
   #  dof                                     basis(begin)   basis(end)
  1   rd                                 1     0.000000327    0.000000357
------------------------------------------------------------------------------
```

Figure 23: The context menu that appears when right-clicking on the text widget. Additional options implemented highlighted in red.

In a similar vein, we can also inherit from `QPlainTextEdit` to provide additional features to the text widget. Adding extra options in the text widget context menu is slightly harder, however. First, the `contextMenuPolicy` attribute must be set to `CustomContextMenu`, which can be done in Qt Designer. This allows us to use the `customContextMenuRequested` signal, which we can connect to our own slot where we can add actions to the menu using the code in listing 15.

```python
def showTextMenu(self, point:QtCore.QPoint):
    text_menu = self.createStandardContextMenu(point)
    text_menu.exec_(text_menu.actions() + additional_actions, self.mapToGlobal(point))
```

Listing 15: Code snippet to activate a custom menu in `CustomTextWidget`.

We first create a `QMenu` using the `createStandardContextMenu` method. Then, to display this menu, the method `exec_` is used, which takes in a list of actions and the `QPoint` (position) to show the menu in the program. We can thus specify any additional options in the `additional_actions` list in listing 15. The `mapToGlobal` method is required to show the menu relative to the top-left corner of the main window rather than the top-left corner of the screen.

The extra options implemented in `CustomTextWidget` are a line wrapping toggling option, which if enabled breaks a long line that goes out of the screen to be displayed over multiple lines using the `setLineWrapMode` method, and a "Save text" option similar to the "Save .npy data" option except we can use the default Python text I/O methods to save the file. Text in the text widget is obtained using `toPlainText()`.

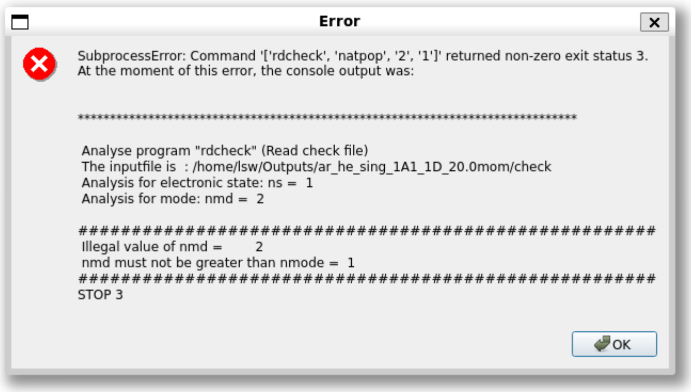
4.5 Running commands using subprocess

So far, we have written analyses for outputs directly from the *Quantics* simulation. In order to interface directly with the Fortran analysis programs we can call them internally in the program. This can be done using Python's `subprocess` module simply using `subprocess.run(args)`, where `args` is a list of arguments, similar to calling the program on the CLI where a space seperates each argument. For example, the CLI command `autospec 9.0 10.5 ev 30 1` is equivalent to `subprocess.run(['autospec', '9.0', '10.5', 'ev', '30', '1'])`. However, doing it this way is not particularly useful as no feedback is returned on the status of the command line process executed. As such, it is useful to set number of optional arguments in the `subprocess.run` method.

Firstly, we must ensure the analysis program is called in the correct directory that the user specifies in the directory widget. This can be done setting the `cwd` parameter to `self.window().dir.cwd`. Secondly, in order to get feedback on the status of the program we must capture its output. In a command line interface, the text that the program outputs is known as the *standard output* (`stdout`). Using the parameters `stdout=subprocess.PIPE` and `text=True` allows the output to be "piped" (sent to) Python in text form (rather than a binary format), such that we can display it in the text widget using `self.window().text.setPlainText(p.stdout)` where `p` is the returned object from `subprocess.run`.

The program may also write to *standard error* (`stderr`) if it encounters an error. This is also an output and can be combined with `stdout` using `stderr=subprocess.STDOUT`. Just because a program writes to `stderr` does not necessarily mean the program failed to work, however: this is instead communicated via the program's *return code*. If the return code is 0, the program terminated successfully; for all other values, it had not. Setting the `check` parameter to `True` allows Python to raise an exception if the return code is non-zero. If these calls to the analysis programs are inside an `AnalysisTab` method, any exceptions raised will be communicated to the user (Fig. 24, also see §4.2 for implementation). However, due to bad programming practices in the Fortran code, calls to the analysis Fortran programs almost always return exit code 0 even if there are errors, with the exception of a couple of cases (e.g. segmentation fault). This ends up causing problems with any analysis methods expecting a certain output from the command line process, causing it to return its own error (Fig. 25). This scenario is futher discussed in the limitations of the GUI (§6.3.2).

By default, the exception message raised by `subprocess.run` is not particularly useful as it only mentions the exit code of the process. The error message from the program is usually in its output which we can append to the exception message by catching it in a try-except block and re-raising the error with the appended message. Similarly, we can catch `FileNotFoundErrors`, which are raised when the specified analysis program does not exist, to append the name of the missing program. To prevent having to do this for each subprocess call, there is a method called `runCmd` in `AnalysisTab` which does all of this well as set the optional parameters as specified above. There is one optional parameter, named `input`, which is passed to `subprocess.run` for analysis programs that require user input (such as `showsys`). The interface with menu-based CLI programs is discussed further in §4.10.



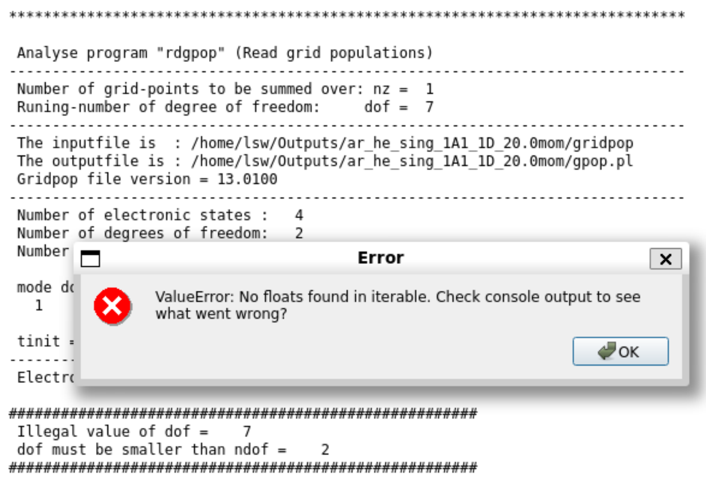
```
Error
SubprocessError: Command '['rdcheck', 'natpop', '2', '1']' returned non-zero exit status 3.
At the moment of this error, the console output was:

********************************************************************************

 Analyse program "rdcheck" (Read check file)
 The inputfile is  : /home/lsw/Outputs/ar_he_sing_1A1_1D_20.0mom/check
 Analysis for electronic state: ns =  1
 Analysis for mode: nmd =  2

##############################################################
 Illegal value of nmd =       2
 nmd must not be greater than nmode =  1
##############################################################
STOP 3
```

Figure 24: The Fortran analysis program returns a non-zero exit code, and the error is successfully captured. Using the `runCmd` method allows `stdout` to be appended to the exception message.



```
********************************************************************************

 Analyse program "rdgpop" (Read grid populations)
--------------------------------------------------------------------------------
 Number of grid-points to be summed over: nz =  1
 Runing-number of degree of freedom:     dof =  7
--------------------------------------------------------------------------------
 The inputfile is  : /home/lsw/Outputs/ar_he_sing_1A1_1D_20.0mom/gridpop
 The outputfile is : /home/lsw/Outputs/ar_he_sing_1A1_1D_20.0mom/gpop.pl
 Gridpop file version = 13.0100
--------------------------------------------------------------------------------
 Number of electronic states :   4
 Number of degrees of freedom:   2
 Number

 mode dc
   1

 tinit =
--------
 Electro

#######################################################
 Illegal value of dof =     7
 dof must be smaller than ndof =    2
#######################################################
```

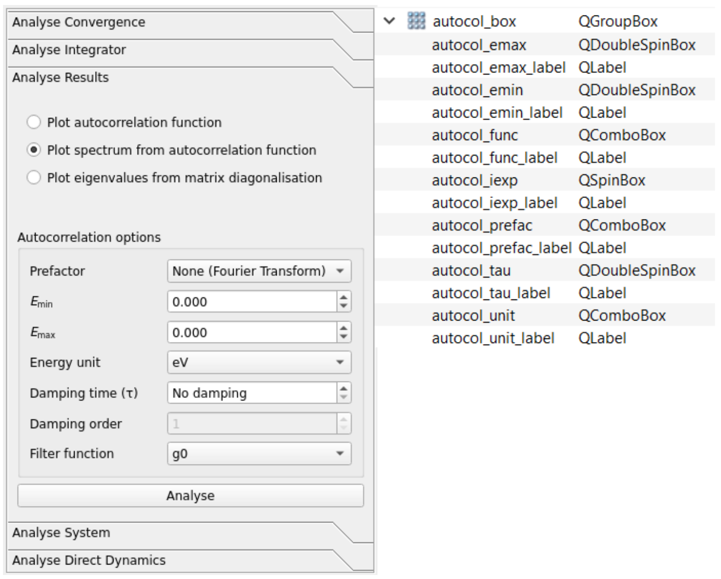
```
Error
ValueError: No floats found in iterable. Check console output to see
what went wrong?
```

Figure 25: The Fortran analysis program returns exit code 0 despite the fact that the program failed to run successfully. The explanatory error is seen in the output (text widget). In this case, it creates an empty file which `readFloats` fails to read—the error "falls through" to the next process.

4.6 Per-analysis options

Most of the CLI analysis programs require additional parameters to control the nature of the output, and these parameters are different for each program. The user must specify these parameters before a call can be made to these programs. To implement this in the GUI, we can create a container full of input widgets in the .ui file, where each input widget adjusts the value of one parameter. In practice, some analysis programs contain so many optional parameters such that it may not be suitable to include them all inside the container. In this case, only the most important parameters can be changed using the input widgets—for any parameters not included, the GUI provides an alternative way to specify these (see §4.13).

Each container is specific to one of the analyses (radio buttons), so a suitable choice of composite widget is `QGroupBox`, as shows a title above the container which we can write which analyses the container corresponds to. Fig. 26 and 27 shows some examples of these `QGroupBox`es.



Analyse Convergence
Analyse Integrator
Analyse Results

( ) Plot autocorrelation function
(●) Plot spectrum from autocorrelation function
( ) Plot eigenvalues from matrix diagonalisation

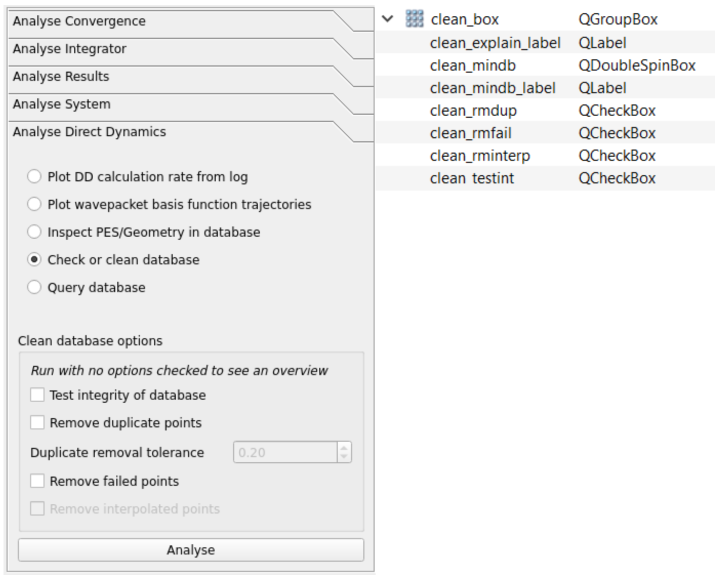
Autocorrelation options

Prefactor	None (Fourier Transform)
E_{min}	0.000
E_{max}	0.000
Energy unit	eV
Damping time (τ)	No damping
Damping order	1
Filter function	g0

Analyse
Analyse System
Analyse Direct Dynamics

autocol_box	QGroupBox
autocol_emax	QDoubleSpinBox
autocol_emax_label	QLabel
autocol_emin	QDoubleSpinBox
autocol_emin_label	QLabel
autocol_func	QComboBox
autocol_func_label	QLabel
autocol_iexp	QSpinBox
autocol_iexp_label	QLabel
autocol_prefac	QComboBox
autocol_prefac_label	QLabel
autocol_tau	QDoubleSpinBox
autocol_tau_label	QLabel
autocol_unit	QComboBox
autocol_unit_label	QLabel

Figure 26: Left: the options container for the `autocol` analysis as it appears in the GUI. Right: the corresponding object list in Qt Designer.



Analyse Convergence
Analyse Integrator
Analyse Results
Analyse System
Analyse Direct Dynamics

( ) Plot DD calculation rate from log
( ) Plot wavepacket basis function trajectories
( ) Inspect PES/Geometry in database
(●) Check or clean database
( ) Query database

Clean database options

Run with no options checked to see an overview
☐ Test integrity of database
☐ Remove duplicate points
Duplicate removal tolerance 0.20
☐ Remove failed points
☐ Remove interpolated points

Analyse

clean_box	QGroupBox
clean_explain_label	QLabel
clean_mindb	QDoubleSpinBox
clean_mindb_label	QLabel
clean_rmdup	QCheckBox
clean_rmfail	QCheckBox
clean_rminterp	QCheckBox
clean_testint	QCheckBox

Figure 27: Left: the options container for the `cleandb` program as it appears in the GUI. Right: the corresponding object list in Qt Designer.

We can then add a container show/hide feature to the `optionSelected` method in the analysis class. If the user passes another dictionary similar to `self.methods` or `self.required_files` (§4.2), this time called `self.options` that contains the name of the container as the value, we can show the parameter container for the analysis that is selected, and only that analysis, using `self.options[index].show()`, and hide all other ones.

By loading the .ui file, the input widgets automatically become attributes of the analysis class, and their values can be directly obtained inside the analysis methods (e.g. using `self.spinbox.value()`, `self.lineedit.text()`, etc.) without needing to add parameters to the function. In the previous sections §4.3.3 and §4.3.4 we mentioned how a combo box can be used to plot different columns. Using what we disscussed above, we can add a `QGroupBox` with a single combo box, then in the method a simple if statement of the form `self.combobox.currentIndex() == value` can be used to plot the user's choice.

The following subsections show examples of using these containers to change the parameters sent to the analysis call.

4.6.1 `checkdb`

The `checkdb` program is not as much of an analysis program as it is a convenience tool. No plotting is performed in this method; information that the program outputs is sent to the text widget. The main purpose of the program is to check the integrity of a QC database (§2.6), but it also has the ability remove failed and duplicate entries. The task that it performs depends on the optional parameters it recieves. As such, it can be used to showcase the parameter container feature of the GUI.

The program will perform a task if it receives the corresponding flag. `-rd` reads the database to test its integrity, `-d` removes any duplicate points, `-c` removes any failed points, and `-sc` removes any failed or interpolated points. Each of these tasks can be represented as a checkbox in the GUI. Using a if statement for each checkbox, if the checkbox is checked (using `.isChecked()`), a string of the corresponding flag is added to a list of optional parameters, which is then sent to `runCmd`.

Since `-sc` is an extension of `-c`, it can be represented as two checkboxes: one for removing failed points, and one for interpolated points. We can then connect a signal that allows the interpolated points checkbox to be enabled only if the failed point checkbox is checked. Then, `-sc` is only added to the list of optional parameters if both the failed and interpolated checkboxes are checked. The code for the connection can be added to the constructor, and the decorator `@QtCore.pyqtSlot` can be used between analysis methods and slot methods in the analysis class.

By default, without any arguments, the program outputs an overview of the database properties, such as which tables are included, the total number of records (points), the number of atoms in the system, etc. To clarify this, a label is appended at the top of the container (see Fig. 27).

The final parameter included in the container is the duplicate removal tolerance, which is how close two points are to one another before they are considered duplicates. This is given the flag `-mindb f` where `f` is a float, so it is represented by a `QDoubleSpinBox` input widget (a `double` is a type of float). As a prerequisite, the flag `-d` must already be given. Like before, we can add a signal to enable the spinbox if and only if the duplicate checkbox is checked. The flag and the value of the spinbox can then be added to the list of optional parameters.

4.6.2 `rdgpop`

For grid-based simulations, if the keyword `gridpop` is given in the *Quantics* input file, a file with the populations of each grid point is given in the `gridpop` file. Unlike the `auto`, `speed` files covered in §4.3, this file is in a binary format and requires an external program to read, namely `rdgpop`. Reading the grid populations is one way to check the accuracy of a MCTDH calculation, as it can show whether enough basis functions have been used. It can also be used to see how much the grid can be made smaller, if there are any unpopulated grid points.

There are two mandatory, positional parameters to this function. The first is the number of grid points to sum over, named `nz`, and the second is the running number of DOFs to consider, named `ndof`. Both of these are integer values and can be represented by a `QSpinBox`. The `runCmd` method can then be called with the argument list `['rdgpop', '-w', str(self.gpop_nz.value()), str(self.gpop_dof.value())]`. The program outputs a file named `gpop.pl` which contains the time-evolution of the grid and basis points; the `-w` flag overwrites any existing files with the same name.

The `gpop.pl` file contains four columns: the start and end of the spacial grid populations, and the start and end of the basis occupations. This file can be read like normal using `readFloats` (akin to listing 12) and a line can be plotted for each column. The `stdout` output, written to the text widget, also contains the maximal values over all times, which may be more useful in some cases.

The plot widget's `pen` parameter in the `plot` method also allows different line types to be used, in addition to the colour of the line, if a dictionary is used instead. In particular, `{'color': 'b', 'style': QtCore.Qt.DashLine}` uses a dashed blue line. This style is used for the "end" columns, while the non-dashed lines are used for the "start" columns.

4.6.3 `qdq`

The `check` file created after a *Quantics* simulation contains various information on the properties of the wavefunction in the propagation, such as the norm, total energy, natural populations (§4.7.1), etc. It also contains data on the expectation values of the coordinate values, $\langle q\rangle$, which can be interpreted as the trajectory of the wavepacket for that coordinate, and the standard deviation $\langle \mathrm{d}q\rangle = \sqrt{\langle q^2\rangle - \langle q\rangle^2}$, which can be interpreted as the width of the wavepacket.

Like in §4.6.2 the `check` file is in binary format and must be read using the `rdcheck` analysis program. Since multiple properties are stored in the `check` file it must know which quantities to retrieve using a parameter, which for this analysis method is `qdq`. The following two positional parameters specify the DOF `ndf` and the state `ns` to retrieve $\langle q\rangle$ and $\langle \mathrm{d}q\rangle$ from. The name of the output file is not fixed and depends on the positional parameters—it is of the form `qdq_{ndf}_{ns}.pl`. If parameters are stored in a list, the file name can be retrieved using an *f*-string `f'qdq_{"_".join(qdq_options)}.pl'`, which are a convenient way to use variables inside strings.

The output file contains three columns with time, $\langle q\rangle$, and $\langle \mathrm{d}q\rangle$, with the latter two being plotted against time.

4.6.4 `autospec`

The absorption or photoelectron spectrum $\sigma(\omega)$ is proportional to the Fourier transform of the autocorrelation function $C(t)$ previously mentioned in §4.3.1. Specifically,[54]

$$\sigma(\omega) = \frac{\omega}{2\pi}\int_{-\infty}^{\infty} C(t)e^{i\omega t}\,\mathrm{d}t = \frac{\omega}{\pi}\int_{0}^{\infty} \mathrm{Re}(C(t)e^{i\omega t})\,\mathrm{d}t \tag{61}$$

However, the integration up to $t = \infty$ is impossible and therefore must end at some finite time T. The actual equation used by the `autospec` program is

$$\frac{1}{\pi}\int_{0}^{T} \mathrm{Re}(C(t)e^{i\omega t})e^{-(t/\tau)^{\mathtt{iexp}}}g(t)\,\mathrm{d}t \tag{62}$$

where the factor $e^{-(t/\tau)^{\text{iexp}}}$ allows the simulate the experimental line broadening and $g(t)$ is a damping function to reduce the effect of the Gibbs phenomenon, which arises from the finite cutoff time. Note that the optional parameter `-EP` multiplies (62) by ω to match (61). By default, `autospec` is set to `-FT` without this multiplication. This choice of prefactor is given in the GUI.

There are five positional parameters required for `autospec`. The first three are the minimum, maximum, and unit of the energy to calculate the spectrum for. The first two are floats and can be represented by a `QDoubleSpinBox` in the GUI. The unit can be any choice of energy unit that is recognised by *Quantics*, of which there are 11: namely, the electronvolt, Hartree (atomic units), nanometre wavelength, kcal/mol, kJ/mol, inverse electronvolt, Kelvin, Debye, megaelectronvolts, millihartrees, and attojoules. All 11 are included as a choice in a combo box.

The label of each item in the unit box does not correspond necessary with the position parameter's name in the GUI. For example, inverse electronvolts have the label eV^{-1} but to use these units the parameter is `invev`. As such, the method defines a dictionary mapping the index of the combo box to the correct parameter name, such that the correct parameter can retrieved using `autocol_unit_map[self.autocol_unit.currentIndex()]`.

The last two parameters specify the damping time τ and damping order `iexp`, respectively. Note that the value $\tau = 0$ is allowed and corresponds to removing the term involving τ completely. To make this more obvious to the user, it is possible to set a special label in Qt Designer if the value is set to 0 which is set to "No damping" in this case (seen in Fig. 26). Additionally, for $\tau = 0$ the `iexp` is not required and can be disabled in the GUI. We can use the damping time box's `valueChanged` signal to check if the damping time is set to 0, in which case the damping order box is disabled. Since the value 0 is "falsy", and all other values are "truthy", the slot can be implemented using a one liner: `self.autocol_iexp.setEnabled(bool(self.autocol_tau.value()))`.

The output of the `autospec` program is to a file named `spectrum.pl`. Along with the energy in the first column, there are three other columns representing equation (62) with different damping functions $g(t)$. By default, the columns correspond to g_0, g_1, g_2 which are

$$g_{n\in\{0,1,2\}}(t) = \cos^n\left(\pi\frac{t}{2T}\right). \tag{63}$$

If the parameter `-lin` is input, the columns correspond to g_3, g_4, g_5, where g_3 is (63) with $n = 3$ and

$$g_4(t) = 1 - \frac{t}{T}, \qquad g_5(t) = \left(1 - \frac{t}{T}\right)\cos\left(\frac{\pi t}{T}\right) + \frac{1}{\pi}\sin\left(\frac{\pi t}{T}\right). \tag{64}$$

All of these functions can be selected in the GUI. If the method ensures that `-lin` is set for $g_{n>2}$ then the correct column can be selected using the slice `[:, i%3+1]` for `i=self.autocol_func.currentIndex()`.

4.7 Plotting a variable number of lines

In this section, we will discuss analysis methods where the number of columns are not fixed, such that the `floats_per_line` parameter cannot be specified. This might happen, for example, where there is a column for a quantity for each state. One can use a simple for loop in this case.

```python
from pyqtgraph import intColor as colr
# ...
for i in range(n_cols-1):
self.window().plot.plot(self.window().data[:, 0], self.window().data[:, i],
                        name=f'col {i}', pen=colr(i, n_cols-1, maxValue=200))
```

Listing 16: Using a for loop to plot a variable number of lines.

A unique colour is used for each line to differentiate between lines. This is done using pyqtgraph's `intColor` method, which we specify the index and the number of hues. If we specify the number of hues as the number of columns (minus the column used for x), then `intColor` will automatically step through a collection of colour values evenly split across the colour spectrum, which can end up making a rainbow effect (see Fig. 28). The `maxValue` specifies the brightness, which is made slightly darker than the default value to contrast against the white background.

4.7.1 `natpop`

The natural populations were previously mentioned in §2.4. They indicate the convergence of the MCTDH calculation. This data is stored in the `check` file, similarly to `qdq` (§4.6.3), for which the corresponding analysis program is `rdcheck`, this time with the `natpop` parameter.

After specifying the mode `nmd` and state `ns` the time-evolution for each SPF in the mode is given in the output file with form `natpop_{nmd}_{ns}.pl`. There is one column for time and one column for each SPF,

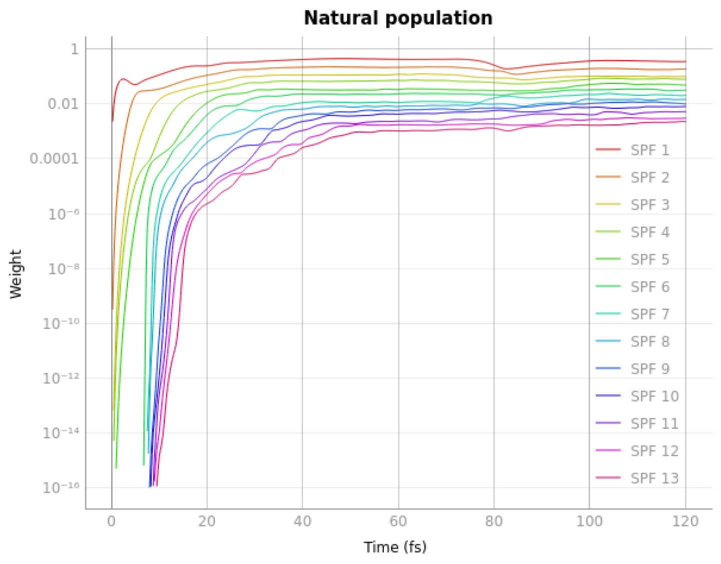


Figure 28: Example of a `natpop` output. A line is plotted for each SPF.

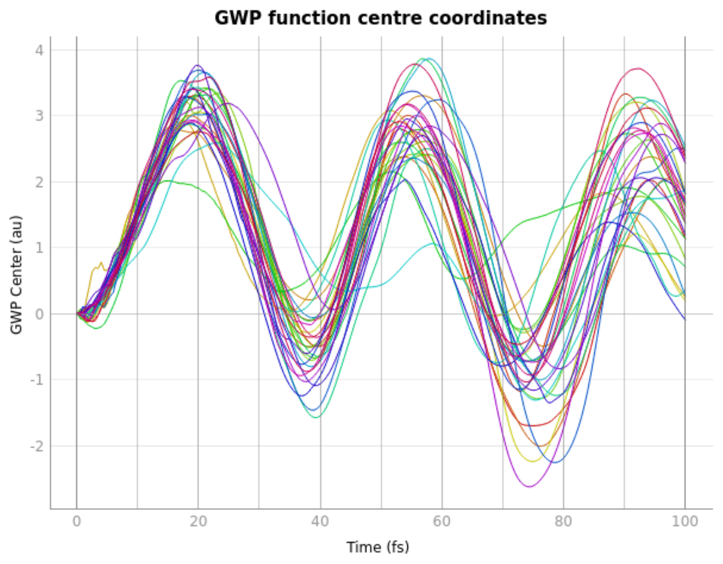


Figure 29: Example of a `gwptraj` output. A line is plotted for each GWP, of which there are 30. The legend is not shown for this analysis.

which can be read using `readFloats` and them plotted using a for loop per listing 16. The standard output (in the text widget) also contains information of the maximum populations over all times for each mode and state, which may also be useful for the user.

4.7.2 `statepop`

For systems with more than one electronic state, the diabatic state populations may be of interest to the user. In the `input` file, electronic states can be specified in a *single-set formulation*, where they are represented as a single DOF where the number of SPFs are set to the number of states, or in a *multiset formulation*, where different sets of SPFs are used for each electronic state: here, the wavefunction is expanded per

$$|\Psi\rangle = \sum_{\alpha} \Psi^{(\alpha)} |\alpha\rangle \tag{65}$$

where, for MCTDH calculations, each state function $\Psi^{(\alpha)}$ is expanded in MCTDH form.

For the MCTDH single-set formulation, the state population is given in the density matrix of the electronic DOF $p_\alpha(t) = \rho_{\alpha\alpha}(t)$, while for the multiset formulation it is given using the norm $||\Psi^{(\alpha)}||^2$.[6] These populations can be retrieved from the `check` file using the program `statepop`. There is also `rdcheck spop` which performs an identical task—the choice of which to use is completely arbitary. No parameters are required for either choice.

The output file for `statepop` is to a file named `spops` containing a time column and one column for each state. The time-evolution of the state population is plotted for each state over `range(1, n_states + 1)`.

4.7.3 `ortho`

The program `ortho` calculates the orthonormality errors of the SPFs. It can be used in conjunction with `norm` to ensure that the norm of the wavefunction remains constant throughout the simulation. However, the `norm` analysis program is currently broken and cannot be interfaced with at the time of writing this dissertation.

No output to a file is made for `ortho`. The output is instead made to `stdout`. This output is rather problematic for `readFloats` as it also contains other information such as the directory of the file, a list of mode labels, etc. If there are any floats in this other information the conversion to a NumPy array will fail. As such, we must read only the grid of values using by selecting the relevant part using regular expressions. This is made convenient as the grid of values are preceded and succeeded by the column headers, which start with #. Using the regex `r'#.*?\n(.*)#'` with the `DOTALL` flag, any text between these headers are selected (see §8.1 for explanation).

There are always three columns in the output: the time, state, and the total orthonormality error. The rest of the columns detail the orthonormality error for each mode. For simulations with multiple states the state column will loop through all the states at each time interval, which essentially makes the data a three-dimensional array with time, mode, and state as the axes. To make plotting easier, the user chooses the state to inspect using the parameter container, such that a multiline plot with a line representing each mode can be used.

To select only the rows corresponding to the user's selection of state, a NumPy mask can be used in the form `self.window().data[self.window().data[:, 1] == state, :]`, where the slice `[:, 1]` selects the state column. If the returned data is empty we can conclude that the user selected an invalid state and an exception is raised. The plotting then proceeds as normal, with each mode being plotted as a seperate line along with the total orthonormality error.

4.7.4 `gwptraj`

For methods using a GWP basis, such as G-MCTDH or vMCG, the trajectories of the centres of each wavepacket can be plotted along a certain mode in position space or momentum space using `gwptraj`. It is normally used as a menu-based program, but using the `-trj` flag, the programs leaves an output file named `trajectory` containing all the trajectories before exiting. A log file, which may contain some useful information to the user, is also created and can be appended manually to the text widget by opening it and using the `self.text.appendPlainText` method.

For the `trajectory` file, `gwptraj` writes a column for time and then, for each GWP, writes a column for each mode. For example, if there are 20 GWPs and 10 modes, the first 10 columns after the time column would be for the first GWP's trajectory for each of the 10 modes, then the second, etc. for a total of 201 columns. `gwptraj` then repeats this for the GWP momenta, for a total of 401 columns.

A visualisation of interest is the trajectory of each GWP along a single mode. If the user wants to look at the GWP centres and specifies the mode to inspect as m, the indices of the correct columns to plot are $m+1, m+n_m+1, m+2n_m+1, \ldots, m+n_{\text{gwp}}n_m+1$ where n_m is the number of modes and n_{gwp} is the number of GWPs. If the user wants to look at the GWP momenta an offset of $(n_{\text{col}}-1)/2$ must be added to m. This can be done using a for loop over `range(offset, offset+ngwp*nmode, nmode)` where the third parameter is the step value or number to increment.

Therefore, we must know either n_m or n_{gwp} to implement this successfully (the other can be calculated from the number of columns). Unfortunately, the `trajectory` does not contain this information by having a “GWP” or “Mode” column (like the “State” column in §4.7.3), but the `input` file does, where the number of GWPs is stated using `ngwp = i`. This can be selected using the regular expression `r'ngwp\s*?=\s*?(\d+)'` where `\s*?` represents any possible number of whitespace before and after the `=` sign. In the unlikely event that `ngwp` is not stated or the input file cannot be found, the user is asked to input `ngwp` manually in a popup using `QtWidgets.QInputDialog.getInt`.

4.8 Bar charts

Categorical data can be visualised using a bar chart instead of a line plot. In pyqtgraph, bar charts can be plotted with a `BarGraphItem`, which can be appended to the plot widget using `addItem`. The parameters for `BarGraphItem` depends on the orientation of the bar chart. For horizontal bar charts, `x` specifies the positions and `height` is set to the values, while in vertical bar charts the extra parameter `y0=0` is required to specify the orientation, then `y` specifies the positions and `width` is set to the values. Unlike in *matplotlib*, pyqtgraph does not have the ability to rotate axis tick labels, meaning a vertical orientation is more suitable for plotting categorical data with long labels while the horizontal orientation is more suitable for histograms (See Fig. 30 and 31).

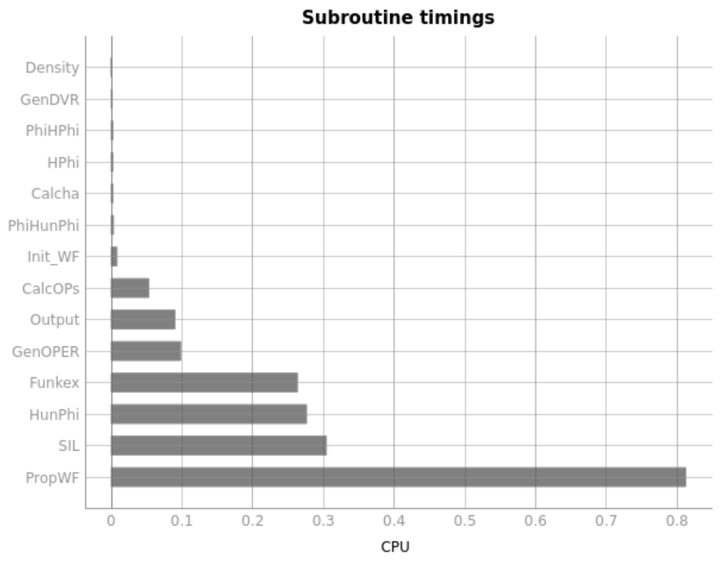


Figure 30: Example of a bar chart visualisation performed in `rdtiming`.

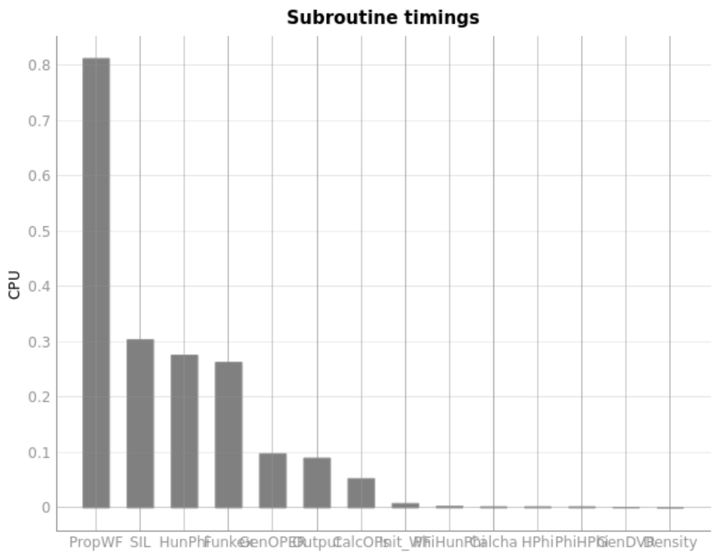


Figure 31: `rdtiming` with horizontal orientation. The tick labels cannot be rotated so this format is unsuitable.

To specify custom tick labels in pyqtgraph, a list of list of tuples must be provided in `AxisItem.setTicks` where the `AxisItem` can be retrieved in the GUI, say for the for the y axis, via `self.window().plot.getAxis('left')`. Each tuple contains the position of the tick, and the label string.

The list of lists has two lists which correspond to major ticks and minor ticks, respectively. Listing 17 contains the code to set the major ticks of a bar chart given a list of positions `positions` and a list of labels `labels`.

```python
ticks = []
for i, label in enumerate(names):
    ticks.append((positions[i], label))
self.window().plot.getAxis('left').setTicks([ticks])
```

Listing 17: Setting custom tick labels in pyqtgraph.

4.8.1 `rdtiming`

The `timing` file is created after a *Quantics* simulation containing the time spent by the computer in each subroutine (function) and the number of times it is called. This can be used to increase the efficiency of a simulation; for example, it can tell the percentage of CPU time that is spent propagating the SPFs. If this percentage is too low or too large, one may consider combining or un-combining modes (§2.5).

The `timing` file contains a column containing the subroutine name and then columns for the number of calls, the CPU time per call, the total CPU time, the percentage of total CPU time spent, and the total clock time. The method `rdtiming` reads this `timing` file and performs two tasks. Given a user selection of column in a combo box `timing_sort`, the method sorts the timing file for that column and then plots a bar chart (e.g. Fig.30).

To do this, the information in `timing` must be parsed. Unfortunately, as the subroutine name is a string the `readFloats` method is unable to read this file. Therefore a special implementation must be used. Firstly, the `timing` file also contains miscellaneous information before and after the grid of subroutine data (such as host/device name, current date and time, and directory). We use regular expressions to split the file into three parts: miscellaneous data before the subroutine data (`pre`), the subroutine data, and the miscellaneous data after (`post`), using `r'(?<=Clock)\n|\n(?=Total)'` where "Clock" is the word before the data and "Total" is the next word after the data (see 8.1 for explanation).

Secondly, the subroutine data is loaded into a table by finding all sequences of characters seperated by whitespace using the regex `r'\S+'` (identical to the EAFP in §4.2.2) in each line. Note a NumPy array is not suitable format for the table as one column contains strings (`numpy.save` still works despite this). However, as the subroutine name may contain spaces, this may make the subroutine name take up more than one entry, but as we know there are 6 columns the leftmost entries can be combined if the length of the row is larger than 6. A final column is added to the column that contains the entire line as a string: this saves having to re-format the data after sorting.

To sort the data, we can use the in-built Python function `sort` with the key parameter specified, which is a function to sort the list by. The key `lambda x: -x[self.timing_sort.currentIndex()]` automatically sorts using the user's specified column. The minus sign reverses the order such that the largest values come first. Note if the user selects the subroutine name as the sort, the minus sign is omitted such that it sorts A–Z. The `timing` file can then be recreated with the sort by using the `pre`, `post`, and the final column of the sorted table.

A bar chart can then be plotted in the same way as described in §4.8. Note if the user selects subroutine name as the column the values of the bar default to the CPU time.

4.9 Animated plots

In the CLI, animated plots are implemented by repeated calls to *gnuplot*, either in a "movie", where a call to *gnuplot* with the updated plot is made a few times a second, or in a step-through plot, where the user hits enter to see the next frame of the animation. The speed of the movie cannot be changed, only the number of frames to increment per step. To finish the animation early, the user must type `Ctrl + C`, and to see a specific frame in the animation, the user must know the index of the frame to view.

The GUI can not only offer the ability to visualise animated plots, but provide a better user experience by allowing customisable speeds, allowing saving an .mp4 movie file of the animated plot, and allowing the user to skip to any frame using a *scrubber*, which is a slider found that controls the timeframe of the animation/movie, usually placed below the video in most media players.

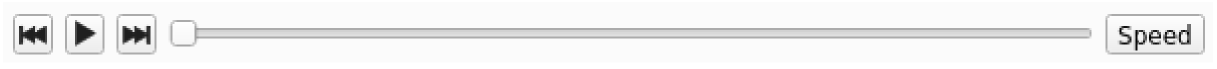


Figure 32: The media widget as it appears in the GUI, which is only visible for animated plots.

The first step in implementing these features is to create a new class inheriting from `QWidget` named `QMediaWidget`. This widget acts at the controller for the time axis in animated plots, and is composed of media buttons (skip to beginning, play/pause, and skip to end), a scrubber (a `QSlider` instance), and a button to control playback speed (Fig. 32). The `QSlider` is a widget for allowing an input within a bound range by moving the handle to a position, changing its interval value. The `valueChanged` signal implemented by `QSlider` can then be connected to a slot that changes the data in the plot widget to the time specified by the handle position. This implementation must be performed per analysis (e.g. see §4.9.1).

A button is embedded for the playback speed instead of a `QDoubleSpinBox` because having the speed shown at all times is not particularly necessary and can become an eyesore. Instead, when the button is pressed, the value appears in a popup which the user can change, using the Qt method `QtWidgets.QInputDialog.getDouble`. This value is stored in an attribute `self.speed` which can be changed by the user using the input dialog (by default it is 30 FPS). Note that this speed is given in frames per second and not propagation time units per second, so simulations with large step sizes may end up being played back faster than expected.

To make the media buttons work they have to be connected to a slot which modifies the value of the `QSlider` (scrubber). For the skip to beginning and skip to end buttons, the slots can be implemented using `self.scrubber.setValue` with `self.scrubber.minimum()` or `self.scrubber.maximum()`, respectively.

For the play/pause button, the attribute `checkable` must be set to `True` in Qt Designer, such that there is a released (click to play) and held (click to pause) state. In the play state, the scrubber should automatically increase its position at the speed given by `self.speed`. To implement this, the button's `clicked` signal needs to be connected to a slot that starts a `QTimer`, which then provides a `timeout` signal that can increment the scrubber value by one every `delta` ms given in `self.timer.start(delta)` (here, `delta = 1000/self.speed`). In the pause state, the timer can simply be stopped using `self.timer.stop()`.

The media widget should only be shown if the user selects an analysis that outputs an animated plot. Therefore, at startup, the media widget is hidden, and only shown if the `animated` parameter is set to `True` in the plot widget's `reset` method, which can be set per analysis.

4.9.1 `showd1d`

The `showd1d` analysis program is used to obtain the system density evolution along a selected DOF and state. This is essentially the time-evolution of the probability density of the wavefunction Ψ^2 after integration over all other DOFs which allow the 1D plot to be made. Interfacing with this program is composed of two parts: retrieving the data from the analysis program, and then setting up the animation using the media widget.

As mentioned, the parameters to this program are the DOF `fx` and the state `sx` which can be selected in the parameter container. This returns a file named `den1d_{fx}` if the state selected is 1, or `den1d_{fx}_{sx}` otherwise. This output file is actually a *gnuplot* command file, and its specific format depends on optional flags: by default, the output is a step-through format `-S` where enter is pressed to plot the next frame, and thus contains multiple lines of *gnuplot* commands. However, we do not need to interface with *gnuplot*, so the most convenient output format is `-T`, a "3D time file", which has only one *gnuplot* call with data for a 3D plot where time is one of the axes. The `-w` flag is also given to overwrite a possible existing file. We can ignore gnuplot commands by passing `'^plot|^set'` to the `ignore_regex` parameter of `readFloats`.

The `-T` flag output contains a grid with four columns: the position along the DOF, the time, the real density and the imaginary density, where the position is incremented for each time interval before time is incremented. Before plotting this, we would like to convert this into a suitable format for an animated plot where the columns for time and densities can be retrieved using a slice to select time. This can be done using the `numpy.split` which can split the array into multiple subarrays. If we assume that the number of rows dedicated to each time interval are the same, we can use `numpy.unique` to retrieve a list of time intervals, which we can then find the size of: this is the number of subarrays we want which can be inputted into `numpy.split`. Now, the real density column for, say, the fourth time interval can be selected using `self.window().data[3][:, 2]`.

Initially, only the lines for the first time interval are plotted. The media widget is shown by setting the `animated` parameter to `True` in `reset`, its scrubber's value set to 0, and its maximum value set to the number of frames (equal to the number of subarrays). To change the plotted values of a line without clearing everything and using `self.window().plot.plot` again, we must first obtain the `PlotDataItem`s, which are the representations of the lines in pyqtgraph's plot widget, using `self.window().plot.listDataItems()`, which return a list of `PlotDataItem`s. The data for the line can then be changed via `PlotDataItem`'s `setData` method.

Thus the implementation is as follows. First, the signal `valueChanged` from the media widget's scrubber is connected to a custom slot. We first ensure it is disconnected from any other slots using `valueChanged.disconnect()`, as these may be from other animated plot analyses which will try to change data that no longer exists as the plot is cleared for replotting. However, this will raise an `TypeError` if there are no connections, so we wrap it in a try-except block, doing nothing if this exception is raised. The slot itself can then retrieve the two lines (real and imaginary density) from the plot and set its data that corresponds to the time interval chosen by the user.

As the time column is still present after the splitting of the array, we can include a subtitle which contains the current propagation time using the `top` label in `CustomPlotWidget`'s `setLabels`. Note that this would normally cause axis ticks and labels to show at the top of the plot screen, but this can be disabled via the custom implementation in `CustomPlotWidget`.

4.9.2 Saving videos

While pyqtgraph's built-in exporting option can save images, it cannot save videos of animated plots as it does not know the mechanism behind the slot that changes its data for each frame. However, since *we* know this mechanism, we can manually implement this ourselves in the `CustomPlotWidget` class. This can be included as an option in the context menu of the GUI (also see §4.4), but should this option should be hidden for non-animated plots, which can be toggled in `CustomPlotWidget`'s `reset` just like toggling the visibility of the media widget.

Firstly, the user selects a savename for the video using `QtWidgets.QFileDialog.getSaveFileName`, similarly to in §4.4. To obtain a video, one can save an image of each frame and then combine it into a video. An image can be generated via code using an `ImageExporter` instance in the `pyqtgraph.exporters` module, which must be imported seperately. For each frame of the plot, the scrubber's value is incremented manually, which will trigger the re-plot. Then, the image gets taken by the exporter, and stored in a temporary directory in the saved file's directory, with its name being the frame index (padded with zeros otherwise by default sorting `40.png` comes before `5.png`, for example). The exact implementation is listed in listing 18.

```python
exporter = pg.exporters.ImageExporter(self.plotItem)
for i in range(self.window().scrubber.minimum(), self.window().scrubber.maximum()+1):
    self.window().scrubber.setSliderPosition(i)
    exporter.export(str(temp_directory/f'{i:05}.png'))
    self.window().scrubber.repaint()
```

Listing 18: Creating an image for each frame of the animated plot. `temp_directory` is the Path object of the temporary directory.

To convert these images to a video, the *FFmpeg*[j] program is used, which is a suite of libraries dedicated to various aspects of handling multimedia files. In particular, we can use this program to concatenate all the images in the temporary directory into a movie. However, we must ensure that the user has *FFmpeg* installed on the command line first. We perform a test call with `subprocess.run` using `ffmpeg -version`, to which if there is an error, an popup appears telling the user the required program is missing.

Otherwise, we can use a console command to generate the video, ensuring that the command is run in the temporary directory. The following parameters are given to *FFmpeg*:

- `-y`: Overwrites an existing video file.
- `-framerate {str(self.window().speed)}`: Framerate of the video, set by the `speed` attribute such that the GUI and video playback have the same framerate.
- `-pattern_type glob`: Specifies the input parameters are given in a *glob*-type format, where the special character `*` matches any string of characters (but does not look in subdirectories).
- `-i '*.png'`: Input files, with pattern-matching as above.
- `-c:v libx264`: Specifies the video codec to use which specifies the encoding for the output file.
- `-pix_fmt yuv420p`: Specifies chroma subsampling of the video, which reduces the file size of the output file.
- `-vf 'pad=ceil(iw/2)*2:ceil(ih/2)*2:color=white'`: The `libx264` video codec requires the input file pixel dimensions to be divisible by 2. If they are not, this parameter pads extra white pixels such that `libx264` can read them.
- Finally, the single positional parameter is the location and name of the output file, which is given by the user. Note that this is passed as an absolute file location, so it will not be saved in the temporary directory.

Afterwards, the temporary directory along with its contents is deleted. The `pathlib` library does not contain a method to delete a directory; the `rmtree` function of `shutil` is used instead. The call to *FFmpeg* is wrapped in a try-except block to raise a custom message if an error occurs with the creation of the video. Otherwise, a confirmation popup is shown telling the user that the operation was successful.

4.10 Coordinate selection

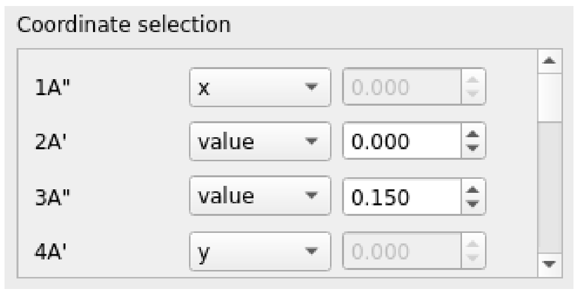
The last program that is integrated with the GUI is `showsys`. It serves as a particular challenge to interface with as it is a menu-based analysis program, where manual input is required after executing the program on the command line rather than using parameters in one command line call. While this would improve the user experience using the CLI, it does not improve developer experience who typically interfaces with subprocesses with parameters only.

An input after the command line call can be given to the `subprocess.run` method. However, only one input can be given, not the multiple that is required to navigate `showsys`' submenus to select options. Thankfully, the saving grace is that multiple inputs can be simulated simply by seperating inputs using a new line `'\n'`. This does mean however, that menu options and parameter format for these options must be known in advance.

[j] `https://ffmpeg.org/`

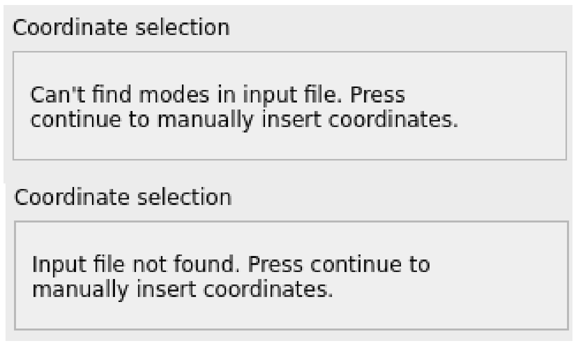
The options that `showsys` can perform include showing the PES of the system and provide 2D density evolutions of the wavefunction. These can be visualised using contour plots, which are discussed in the next section §4.11. However, in order to do so, the choice of which coordinates (DOFs) that represent the two axes are required. These parameters are selected using option 20 in the menu (listing 2), in which one DOF can be selected as x, another DOF can be selected as y, and other DOFs can be set to a specific value. However, unlike in `showd1d` where the index of the DOF is specified, the *mode label*, which provides the name of the DOF in the `input` file, must be specified.

This entire section is dedicated to the creation of a special widget named `CoordinateSelector` which can be embedded in a parameter container. The `input` file is searched to obtain mode labels. For each mode label there is the option to choose between x, y, and "value", for which a spinbox is enabled to enter a value in (Fig. 33).


Coordinate selection

1A"	x	0.000
2A'	value	0.000
3A"	value	0.150
4A'	y	0.000

Figure 33: Example of a `CoordinateSelector` being used in a parameter container.


Coordinate selection

Can't find modes in input file. Press continue to manually insert coordinates.

Coordinate selection

Input file not found. Press continue to manually insert coordinates.

Figure 34: Special messages in `CoordinateSelector` that occur when mode labels cannot be found, or when the `input` file does not exist.

4.10.1 Finding mode labels

The first task is to implement a `findModeLabels` method that can obtain mode labels from the `input` file. Unfortunately, mode labels can be found in different sections, depending on the nature of the simulation. For MCTDH, mode labels are generally found in the `SPF-BASIS-SECTION` (or `SBASIS-SECTION` in older `input` files), and for DD-vMCG, mode labels are *sometimes* found in the `nmode` subsection of a `INITIAL-GEOMETRY-SECTION` or the `DD-GB-SECTION`. Information in these sections can be obtained using regular expressions as the sections end with the phrase `END-` then the name of the section.

As an example, information in the `SPF-BASIS-` or `SBASIS-SECTION` is selected using `r'S(?:PF-)?BASIS-SECTION\s*\n(.*)\nend-s(?:pf-)?basis-section'` with the `re.DOTALL` flag. Mode labels here are written in the form `mode_label1 , mode_label2 , ... = nspf1 , nspf2 , ...` where `nspf` is a digit (or sometimes the special keyword `id`). Another regex is used to select characters before the equals sign, and have the characters split by a comma (using `.split(',')`) and whitespace removed (using `.strip()`) in a list comprehension. Note that other keywords may be defined here such as `packets` or `gwp-type` which we ignore as these are not mode labels. For the `nmode` subsection each mode label is the first sequence of characters on a new line. Explanations for all regexes are found in §8.1.

While the implementation above can cover a wide range of cases, it is unfortunately not exhaustive. For example, for ML-MCTDH simulations, mode labels are given in an `ML-BASIS-SECTION` using a tree structure (Fig. 7). For some DD-vMCG simulations no mode labels are given in the `input`, in which they are read from the database file. Special parameters such as `elcont` given in other sections change the name of certain mode labels (for this case `el` is changed to `Elcont`), and so on. These particular limitations are discussed further in §6.3.3. If an `input` file exists but no mode labels can be found using the above implementation an empty list is returned from `findModeLabels` which can be detected and cause a special message to appear (Fig. 34).

4.10.2 Adding subwidgets

Assuming we have a list of mode labels however, we can now build the widget around this. We must manually create the widget using code instead of in Qt Designer as the number of mode labels is variable. In the method `addModeLabels`, a subwidget is created for each mode label that includes the `QLabel` for the label, the combo box to select x, y, or "value" from, and the spinbox to choose the value. If there is only one label, x is given as the only option in the combo box. Otherwise, by default the first two labels are chosen to be x and y, and the rest are set to "value" using `select.setCurrentIndex(min(i, 2))` where `select` is the name of the combo box. For the first two labels, the spin box is also disabled accordingly.

We also need to connect signals to these subwidgets so they behave as expected. In particular, the spinbox should only be enabled if "value" is selected, and only one DOF can be chosen as x and y. In the `addModeLabels` method, each combo box is connected to a slot named `selectChanged` for this purpose. When this slot is triggered, firstly, we need to know which subwidget the combo box that had been changed is in. This can be done with the aid of PyQt's `self.sender()` method, such that the index can be obtained using `self.layout().indexOf(self.sender().parent())`, which allows us to find the specific spinbox to enable or disable. Secondly, if the user picks x or y, then any x or y in other subwidgets need to be changed to "value". We iterate through all the subwidgets in the coordinate selector and change the combo box to "value" if they match the index selected, using `self.layout().itemAt(i).widget()` to obtain the subwidget attribute. The subwidget index that `self.sender()` is in is skipped, of course.

We will also need to provide a `clearWidget` method. If the user changes directory with a different set of mode labels, the subwidgets in the coordinate selector need to be cleared before new ones are added, done by iterating through all subwidgets and using the `deleteLater` method. Then, a `refresh` method to combine clearing the widget and adding new subwidgets with `addModeLabels`.

4.10.3 An aside: Splitting the `AnalysisTab` constructor

In order to integrate this with an analysis method, the `optionSelected` in the analysis class can be overridden to refresh the coordinate selector when the radio option containing the widget is selected. This allows the widget to generate its subwidgets on-demand rather than during its intialisation. Because `optionSelected` is called when any of the analysis radio buttons are selected, in order to ensure that the options inside the selector do not reset when selecting another radio button and returning, the current mode labels are saved in an attribute `self.mode_labels`. If `findModeLabels` returns an identical list, `refresh` does not need to do anything.

However, we also want to refresh the widget when the user changes the directory. This can be done by connecting the `textChanged` signal in of the text edit in the directory widget to the `optionSelected` slot also. Alternatively, it can be connected to `refresh` directly, but it is better to be general if other custom widgets similar to coordinate selector are implemented in the future. However, since accessing the directory widget requires the pointer to the main window, which is given via `.window()`, there is a problem when we add the connection in the analysis class constructor. The `.window()` method does not work in `__init__`, instead returning a pointer to itself, which means that any attempted connections made in the constructor do not work (as the analysis tab does not contain a directory widget itself). This is also a problem when implementations of `runCmd` and `checkFileExists` depend on menu options in the main window (§4.13).

The reason why `.window()` does not work in `__init__` is because `.window()` recursively finds the parent object of the widget until it reaches the top level widget (main window). When a widget such as `CoordinateSelector` is initialised, the children and parents of the widget are not set until the end of `__init__`. Solving this issue requires splitting the constructor for the `AnalysisTab` class in two. We can delay creating connections until the final constructor is called, which is in the `AnalysisMain` class. We create a method named `activate` containing everything previously in the constructor except loading the .ui file, meaning that connections and dictionaries mentioned in §4.2 are given in `activate` instead. Then, in the constructor of the `AnalysisMain` class, `activate` is called for all the analysis classes to provide full functionality.

4.10.4 Using coordinate selector in analysis methods

The premise of the `CoordinateSelector` widget is to provide the parameters required in the "Change coordinate selection" submenu of the `showsys` command line program. On each line in this menu, the user types the mode label, followed by `x`, `y` or a constant number. We can create a method `__str__` to convert the user's choices in the GUI to the text that the user would type in the coordinate selection submenu. This can then be used in the analysis method as part of the `input` parameter to `runCmd`. Implementing `__str__` involves simply iterating through all subwidgets, then adding its mode label and either `x` or `y` from the combo box or the value from the spinbox, before attaching a new line at the end. Note that an error occurs when no DOF is selected as x in the CLI: this is replicated in the `__str__` method also. If `findModeLabels` did not return any mode labels, then `__str__` will return an empty string. This can be identified in the analysis program and have an alternative implementation as necessary.

In addition, a similar implementation can be used to obtain the mode label for chosen as x or y. This is particularly useful as we can use this to include the mode label as axis labels during plotting.

4.11 Contour plots

In the previous section, we developed a widget that allows the user to select the values for coordinate to use for the x and y axes. Using `showsys`, this will produce a 3D contour plot with the contours representing the values on the z axis. Unfortunately, as mentioned in §4.3, pyqtgraph does not have a built-in method to produce a contour plot as of the time of writing this dissertation. It does however contain a method to produce an *isocurve*, which is a single contour line, which we can use to add a `plotContours` method in our `CustomPlotWidget`.

The pyqtgraph `IsocurveItem` has two main parameters. The first is a matrix which contain the values of z for each coordinate in x and y. The second is the level of the isocurve, which corresponds the value of z that the isocurve is drawn on. The contour plot can then be implemented using a for loop over a range of levels. However, note that the `IsocurveItem` does not have `x` or `y` as parameters. Instead, the bounds of the data array `z` are implicitly assumed to be the indices of the array (e.g. passing a 30×30 array plots it over (1, 1)–(30, 30)). To fix this, we can use PyQt's `QtGui.QTransform` class to define a translation and scale to apply to the isocurve, assuming that the values in `x` and `y` are linearly spaced (Listing 19).

```
tr = QtGui.QTransform()
tr.translate(x.min(), y.min())
tr.scale((x.max() - x.min()) / np.shape(z)[0], (y.max() - y.min()) / np.shape(z)[1])
for i in range(len(levels)):
    c = pg.IsocurveItem(data=z, level=levels[i], pen=colours[i])
    c.setTransform(tr)
    self.getPlotItem().addItem(c) # add to plot widget
```

Listing 19: Code snippet for transforming the isocurve to align with the arrays `x` and `y`. `levels` and `colours` are the list of levels and line colours for each isocurve.

We can then create a method `plotContours` based on listing 19 that has `x`, `y` arrays, and the matrix `z` as parameters where the length of `x` and `y` match the shape of `z`. Then we can add an additional parameter `levels` which can be sepcified either with a list of z-values to draw isocurves over, or an integer for the size of list which is generated from equally spaced values between `z.min()` and `z.max()`.

Lastly, it would also be useful to include a colourbar, such that an isocurve having a particular colour on the colourbar represents that value of z. There is a convenience function in pyqtgraph that allows this using `self.getPlotItem().addColorBar`, but is meant to be coupled with an image plot, which is a required parameter. We do not actually want to include an image plot over the contours, so we can simply pass an empty image map `pyqtgraph.ImageItem()` into the parameter. This colourbar can be hidden at startup and in `AnalysisTab.reset` and is only shown when visualising a contour plot using the `plotContours` method.

The colourbar also requires a colour map, which defines the colour for each value in the colour bar. One can use the `pyqtgraph.colormap.get` to select from a default collection included with pyqtgraph—for this GUI, the "CET-R4" colour map was used. Then, in the implementation for `plotContours`, a list of colours can be obtained from the list of levels using `self.colourmap.getLookupTable` with parameters `nPts=len(levels), mode=pyqtgraph.ColorMap.QCOLOR`. To ensure the numerical tick labels are correct, the colour bar's `setLevels` method can be used with `z.min()` and `z.max()` as parameters.

4.11.1 `showpes`

One of the tasks that `showsys` can perform is to plot the PES of the system. If we would like to only plot the PES, the flag `-pes` can be added as a parameter; this will save time that is otherwise spent reading the `psi` file. Normally when using the CLI, the user would directly call *gnuplot* using the "plot to screen" option (Option 1, listing 2). To obtain data in a file, the "save data to an xyz file" option can be used (Option 5).

The `showpes` method, which calls the `showsys` program to obtain the PES, has parameters to show the either the diabatic and adiabatic PES, the state to select, and the coordinates provided by the coordinate selector (§4.10). These correspond to option 10, "change plot task", option 60, "change state selection", and option 20 respectively. As mentioned before, these parameters are no longer given to the call to the command line but using the `input` parameter, which must a string containing the keystrokes required as if the Python code was simulating a user selecting these options on a CLI. For example, to select the task of plotting the diabatic PES, one types 10, enter, 2 to select the correct option "plot diabatic pes", enter. So the input string is `'10\n2\n'`. A similar implementation is used to select the state.

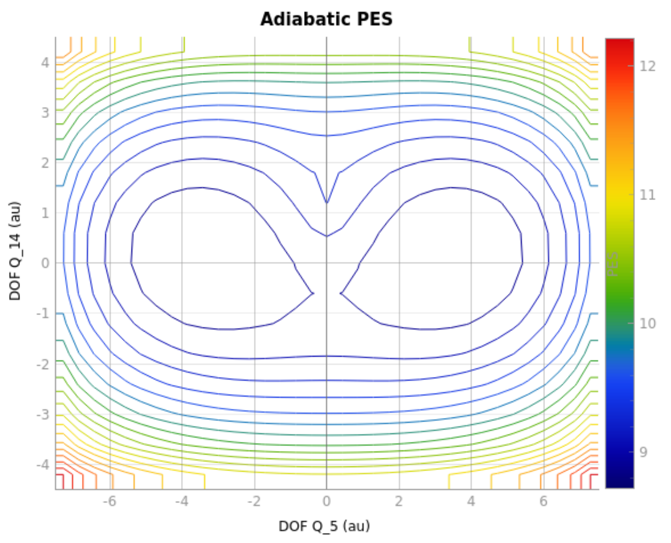


Figure 35: Contour plot of the PES of the lower excited state of butatriene, as shown in the GUI. Compare Fig. 8.

The keystrokes required to select the coordinates can be produced with the aid of `CoordinateSelector.__str__`, such that the input string is simply `f'20\n{str(self.showpes_coord)}\n'`, where `self.showpes_coord` is the coordinate selector instance in the `showpes` parameter container. However, in case the coordinate selector did not manage to obtain the mode labels, QtWidgets.QInputDialog.getMultiLineText is used to obtain the input, where the experience is identical to writing down the coordinates manually in the CLI except the intermediate output containing the list of mode labels to choose from is not included.

Finally, the "save data to an xyz file" option is selected using 5, enter, and the query for the saved filename of the output is entered as pes.xyz. 0 must then be typed to exit the program, otherwise an error occurs as the program is terminated while still waiting for user input. The inputs for interacting with each submenu are concatenated before the command is run. This produces the output PES file which can be read. showsys also writes a .log file of details; since the standard output is not very illuminating (just lists and lists of options), the text in the text widget is replaced with the contents of the log file.

If the user did not select a y value, columns for a 1D line plot are output, which can be parsed and plotted as normal per listing 12. Otherwise, the PES file contains columns for x, y, and z. x coordinates are incremented for each y value, then y values are incremented, producing a z value at each coordinate grid. The column format for z is unfortunately not the format that is required for plotContours, which require a grid of z values. As such, a conversion is necessary, where the 1D z column is reshaped into a 2D matrix with the size of the x and y arrays (listing 20).

```
x = np.unique(self.window().data[:, 0])
y = np.unique(self.window().data[:, 1])
z = np.array(self.window().data[:, 2]).reshape(y.shape[0], x.shape[0]).T
```

Listing 20: Conversion from the data in a .xyz file to x and y arrays and a z matrix.

These values can then be used as parameters for plotContours. The xcoord and ycoord attributes of the coordinate selector can be used in the labels for the resulting plot. An example can be seen in Fig. 35.

4.11.2 showd2d

Another task that showsys can perform is to plot the 2D reduced density evolution, i.e. the 2D equivalent of showd1d (§4.9.1). There is also a non-menu based program that performs the same task known as dengen, but was not interfaced with as dengen does not produce the correct output, either crashing due to a segmentation fault or producing an output file with only NaNs.

The parameter -nopes can be passed to save time reading the oper file. The showd2d analysis method output will be an animated contour plot, combining implementations for animated plots (§4.9) and contour plots (§4.11). The underlying data plot will have x, y, z, and t values, making it essentially a 4D array. Here, no parameter is needed to select adiabatic or diabatic regimes; the parameters are for the state and coordinate selection only. Otherwise, the creation of the input string is identical to §4.11.1. The case where a y coordinate is not selected does not have to be handled as an information popup can be created telling the user to use showd1d instead.

The difficult part of implementing this method is due to the format of the output .xyz file, as there is no column for time as there was in showd1d. Instead, the format is identical to the previous section with columns for x, y, and z, but with two empty lines between each time interval, before the x, y, and z columns for the next time interval is included. These two empty lines can be matched using the regular expression `r'(?:\n\s*){3}'`. The `\s*` is required as one of the blank lines contain whitespace so matching three consecutive new lines does not work.

Therefore, to create the data array for this analysis the entire output file must be read, then split into individual xyz blocks using the re.split function with the regex provided above. Each of the resulting blocks can then be converted into a grid format using listing 20 and added to a list; however, the x and y arrays only need to be created using the data from the first time interval, as we assume that x and y are the same for all time intervals. The resulting list of NumPy arrays is then converted into one big 3D NumPy array that represents the time evolution of the z-grid.

Afterwards, the media scrubber's valueChanged is disconnected and reconnected as per the implementation in showd1d. For the slot's implementation, the use of self.window().plot.listDataItems() does not work as this only retrieves PlotDataItems, not IsocurveItems. These can instead be retrieved using self.window().plot.getPlotItem().items, which is a list of all objects in the plot, which for a contour plot only consists of IsocurveItems. The data for each isocurve is then updated using .setData with the corresponding z data for the time interval representing the scrubber's position.

Finally, when plotting the contour plot for the first time interval, the levels parameter should be a list containing the minimum and maximum levels of z over all time intervals to get a better representation of the true range of values, as otherwise only contour levels between the minimum and maximum of the first time interval are shown.

4.12 New analysis functions

As mentioned in §1.3, the analysis GUI also contains a handful of analysis options not previously implemented in Fortran. These new analyses are all in the topic of DD-vMCG (§2.6), which is comparatively a much newer addition to the *Quantics* package, and are written purely in Python.

At the moment, these analyses are simply methods in an analysis class that are called from the selection of the relevant radio button (§4.2). Should future work on these Python analyses make the file size too large, individual analyses could potentially be moved to a new file and parameters added to them. These files can simply be imported and the method in the analysis class call them with parameters selected by the user in the GUI.

4.12.1 `calcrate`

In a DD-vMCG calculation, any calls to external quantum chemistry programs such as *Gaussian* or *Molpro* (§2.6) are recorded in the `log` file. The `calcrate` method reads this file and plots how many calls are made at each time interval. The part of the `log` file we are interested in is in the form of listing 21.

```
 time[fs]       24.000
DD Update requires new point at:    24.229195
No. QC calculations :     1
Generating DB Point for GWP :     1
 Generating symmetry replica :     2
Propagating diabatic states to DB record:          3251   from: 3213
Propagating diabatic states to DB record:          3252   from: 3214
DD Update requires new point at:    24.229195
No. QC calculations :     1
Generating DB Point for GWP :    25
 Generating symmetry replica :     2
Propagating diabatic states to DB record:          3253   from: 787
Propagating diabatic states to DB record:          3254   from: 788
DD Update requires new point at:    24.316975
```

Listing 21: A text snippet of recorded calls to QC programs in a `log` file.

The algorithm is as follows. The file is read, and for each line, if it contains the sequence "`time[fs]`", the time is appended to the list `times` using the regex `r'[+-]?\d+(?:\.\d*)?'` (This is a part of the float regex, explained in §8.1). Then, a new entry is appended to the list `n_calcs` initialised to 0. If it contains the sequence "`No. QC calculations`", the following number is added to the last entry of the `n_calcs` list. `n_calcs` can then be plotted against `times`. The contents of the `log` file are also written to the text widget, for good measure.

4.12.2 `querydb`

As the QC database is implemented using SQLite, it can be queried using SQL. The `querydb` allows the user to use any SQL query to inspect the QC database and have the results returned in a formatted table in the text widget. To connect to the QC database one can use the `sqlite3` package, part of the Python standard library, and to execute a query one can use code in the form of listing 22.

```python
con = sqlite3.connect('file:/home/user/sim/dd_data/database.sql?mode=ro', uri=True)
cur = con.cursor()
res = cur.execute(query).fetchall() # can repeat for multiple queries
con.close()
```

Listing 22: Code snippet for connecting to an SQLite database and executing an SQL query named `query`.

The QC database is always called `database.sql`. The mode `ro` opens the database in read-only mode, such that any write queries generate an exception. A checkbox is given to the user to indicate whether they want to disable read only mode—if this is checked, the database is opened using mode `rw` (read-write). The user is then given a `QPlainTextEdit`, which can recieve multiline strings (this is the same class the text widget inherits from, except with editing enabled), to input an SQL query, which is executed using listing 22. For long SQL queries, the height of the `QPlainTextEdit` automatically adjusts itself such that all of the query is visible using the `sqlChanged` slot.

The returned result from the query is a table in the form of a list of tuples, where each tuple represents a row. To display this we can use the text widget, but simply using `str` to convert this table into a string is not very helpful as it is not nicely formatted—each cell in the table may have a variable size so columns are not necessarily aligned. To fix this we can create a method in `CustomTextWidget` named `writeTable`.

This method uses the Python `format` method to pad cells to a fixed size. For example, `'{:>15}'.format(string)` pads a string to 15 characters aligned to the right, and `'{: .8e}'.format(num)` formats a float to always use scientific notation using 8 decimal places, and prepend a space for positive numbers (as negative numbers require an additional character). It also has other parameters such as `colwidth` to vary the cell size, whose implementation requires formatting the formatter which can be done using an *f*-string. At the moment, the `CustomTextWidget.writeTable` method is used only for `querydb`, but in the future, new analyses using Python may find it useful.

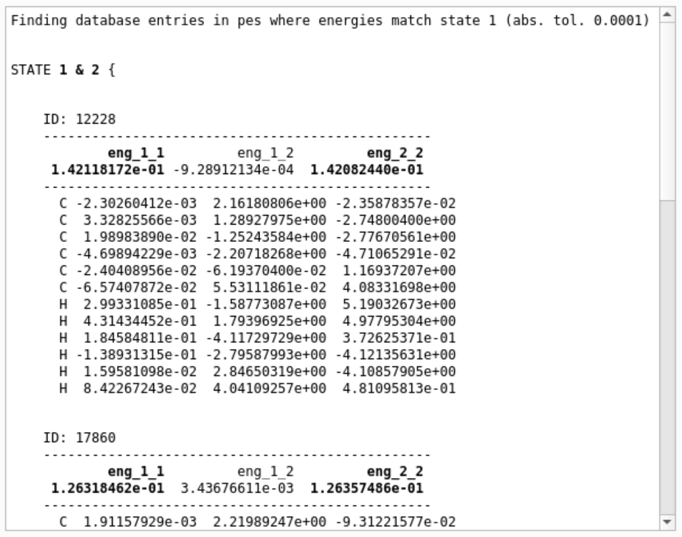
4.12.3 `ddpesgeo`

The final analysis program, unconvincingly named `ddpesgeo`, is dedicated to finding the geometries of the molecular that correspond to certain energies in the PES of a QC database. This can be used by the user to find particular molecular geometries corresponding to minima or conical intersections in the PES (discussed in §2.6). The user can choose between the diabatic and adiabatic potential energy surfaces using a combo box, and can select geometries between two energies, in which a search is performed over all states, or geometries where the energies between two states coincide within a specified tolerance, in which the user specifies the first state and a search is performed over the second state[k]. This choice is given via two radio boxes. As the parameters are different for each option, the radio box is connected to a signal which shows/hides two parameter subcontainers with a similar implementation to §4.6.

The specific implementation is dependent on the version of the QC database. The tables present and column names for each table are different for each version, of which the latest version is version 4. This value is found in the `dbversion` column of the `versions` table. At the moment, `ddpesgeo` is only configured to work with version 4: for other versions an error is returned to the user.

The energies for state `s` are located in the `eng_{s}` column for the `apes` table and `eng_{s}_{s}` column for the `pes` table (nonidentical numbers refer to couplings). A dictionary named `state_name` is created mapping state number to column name for convenience. To perform a search over all states the number of states must be known; this value is found in the `nroot` column of the `refdb` table. Then, for each state, a query is made to the database. The values for the energy are located in a different table than the geometries, so the `pes` or `apes` and `geo` tables must be combined with a *join*; since both tables contain the column `id` (which the join is performed over), tables can be merged together (temporarily) if two records share the same `id`.

For the energy between intervals task, the query is `SELECT * FROM @table LEFT JOIN geo USING(id) WHERE @state_name[s] BETWEEN @emin AND @emax;` where the `@` indicate variables inputted in Python using *f*-strings and the ∗ selects all columns. For the matching energies task, the query is `SELECT * FROM @table LEFT JOIN geo USING(id) WHERE ABS(@state_name[s2] - @state_name[s1]) <= @tol;`.



```
Finding database entries in pes where energies match state 1 (abs. tol. 0.0001)

STATE 1 & 2 {

    ID: 12228
    ---------------------------------------------
           eng_1_1         eng_1_2         eng_2_2
     1.42118172e-01 -9.28912134e-04  1.42082440e-01
    ---------------------------------------------
      C -2.30260412e-03  2.16180806e+00 -2.35878357e-02
      C  3.32825566e-03  1.28927975e+00 -2.74800400e+00
      C  1.98983890e-02 -1.25243584e+00 -2.77670561e+00
      C -4.69894229e-03 -2.20718268e+00 -4.71065291e-02
      C -2.40408956e-02 -6.19370400e-02  1.16937207e+00
      C -6.57407872e-02  5.53111861e-02  4.08331698e+00
      H  2.99331085e-01 -1.58773087e+00  5.19032673e+00
      H  4.31434452e-01  1.79396925e+00  4.97795304e+00
      H  1.84584811e-01 -4.11729729e+00  3.72625371e-01
      H -1.38931315e-01 -2.79587993e+00 -4.12135631e+00
      H  1.59581098e-02  2.84650319e+00 -4.10857905e+00
      H  8.42267243e-02  4.04109257e+00  4.81095813e-01


    ID: 17860
    ---------------------------------------------
           eng_1_1         eng_1_2         eng_2_2
     1.26318462e-01  3.43676611e-03  1.26357486e-01
    ---------------------------------------------
      C  1.91157929e-03  2.21989247e+00 -9.31221577e-02
```

Figure 36: Example output of a `ddpesgeo` analysis with the matching energies task.

The results can then be written to the text widget inline inside the for loop, but to split responsibilities of querying the database and formatting the results, an intermediate dictionary `pesgeo` is created to store the results of each query. The dictionary has the state or states representing the key using a `frozenset` (an immutable set; here it only contains one or two items) and a list of entries where each entry is also a dictionary containing the ID, energies, and a NumPy array of the geometries, whose conversion is made simple using the `geo` table column names, which are `x_{n}`, `y_{n}`, and `z_{n}` for each atom `n`.

The `pesgeo` dictionary is then formatted per Fig. 36. The particular format of the geometries shown is in a format that can be read by a program such as *Jmol*[l] which can view the structure of the molecule. Firstly, the selected entries are grouped by their state, or pairs of state if the task is to find matching energies, using two curly braces `{...}`. This allows the user to select any one particular state or pairs of states more easily. Secondly, the ID and energies of the matching entry are printed. The column names of the `pes` or `apes` table can be retrieved using SQLite's special function `pragma_table_info`, from which the names can be selected using `SELECT name FROM pragma_table_info("@table");`. Thirdly, the geometries can be formatted nicely using the NumPy method `np.array2string`, passing a formatter such as the ones mentioned

[k]Matches over more than two states are not implemented as this is very uncommon and performing the search is more computationally costly.

[l]`https://jmol.sourceforge.net/`

in §4.12.2. Lastly, we can highlight the matching energies if we use `QPlainTextEdit`'s `appendHtml` method instead of `setText` to obtain HTML formatting capabilities, namely allowing bold text via the `<b></b>` tag. This does mean normal line breaks do not work anymore. For some reason, breaking a line with `<br/>` does not work: instead, each line must be wrapped in a `<pre></pre>` tag before `<br/>` can be used.

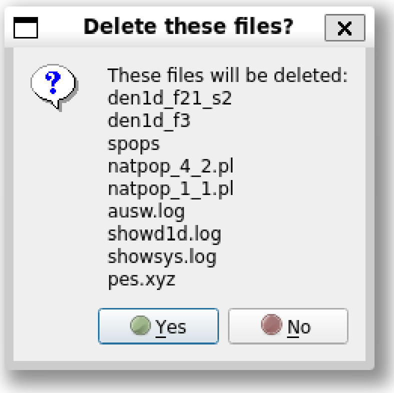
4.13 Additional options

There are a number of additional options that can be selected in the "Options" menu, located in the menubar, that modify the behaviour of the GUI, intended to give more advanced users more control over certain aspects of the program. The first option is a *timeout* option, which controls the time to wait for a command line program to return an output before giving up (killing the process), as well as the maximum time a SQL query can take. This value has no influence on how much time is allowed for the Python GUI to read files or plot the output, however.

Using a `QWidgetAction` to embed a `QDoubleSpinBox` in the menu (like in §4.4), the user can select the timeout limit and the value can be accessed via `self.window().timeout.value()`, which can be input into the `timeout` parameter in `subprocess.run` and in `sqlite3.connect`.

As calls to most Fortran analysis programs leave behind a file, performing many analyses in the same directory will clutter the directory with many output files. These files are left behind after an analysis as generally, a file should not be deleted without the user's permission. Instead, the user is given the "cleanup directory" option, which allows them to delete all possible analysis output files in one click. To implement this, a list of patterns to match is given in a *glob*-type format, where, for example, the glob pattern `qdq_*.pl` matches all possible `qdq` output files for any DOF and state (§4.6.3). For each pattern `glob`, a list of analysis output files that exist in the directory `files` can be produced using `files.extend(list(self.dir.cwd.glob(glob)))`.

The names of each file found is then given in a confirmation question using `QtWidgets.QMessageBox.question` (Fig. 37). If the user selects "Yes", each file will be deleted using `file.unlink()`, and a confirmation popup is given. Otherwise, the operation is cancelled. If `files` is empty, a popup indicating no files were found is shown.



These files will be deleted:
den1d_f21_s2
den1d_f3
spops
natpop_4_2.pl
natpop_1_1.pl
ausw.log
showd1d.log
showsys.log
pes.xyz

Figure 37: Example of the confirmation popup that appears when cleaning up output files.

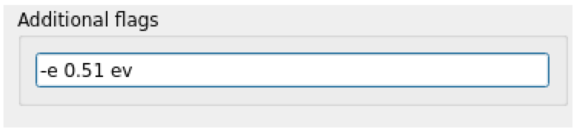

Additional flags: -e 0.51 ev

Figure 38: An additional `QGroupBox` that appears when the "Allow additional flags" option is selected. Here, the flag `-e` tells `autospec` to shift the energies by this amount.

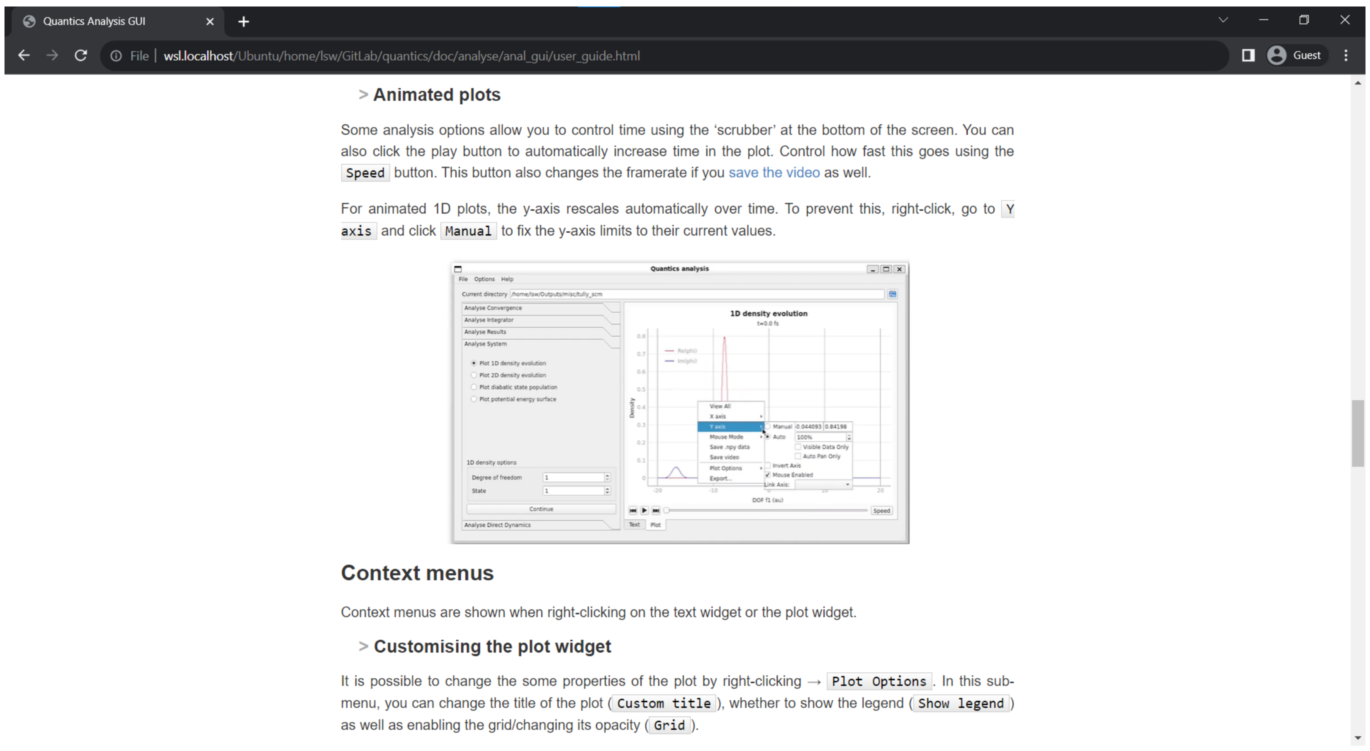
As previously discussed in §4.6, some analysis programs have many optional parameters, some of which are only used in niche cases. As such, these options are not included in the parameter containers, but the advanced user is still free to specify them by including them in the "Additional flags box". This box is normally hidden, but can be enabled via the options menu. On any call to the `runCmd` method, flags input into the additional flags box are split into individual parameters using `shlex.split`, then inserted into the arguments list. The flags must be inserted before the positional parameters since it appears that Fortran stops reading any further arguments if all the positional parameters are received. This can be done using `args[1:1] = add_flags`.

One of the parameters the user can input into the additional flags box can change the required files that an analysis method needs, namely, for analysis programs that read the `psi` file, the flag `-r` or `-rst` reads the `restart` file instead, which contains the information (including contents of `psi`) to continue a simulation if it had been stopped abruptly, or the user wants to extend the propagation time. As such, the implementation that disables the "Analyse" button must be disabled if the user.

The final option given is the option to disable any calls to the Fortran analysis programs completely, named "No command mode". The reason why someone may do this is if they have the analysis output from the command line program already, and they only want the GUI to plot this output, without needing to call the program again. This is particularly useful if the analysis takes a long time (e.g. 2D contour plots of the PES in a DD-vMCG simulation). In this case, a manual call on the CLI can be made, and run in the background without needing the GUI to be open all the time, or the analysis can be performed on an HPC cluster and only the output file needs to be transmitted back to the user. When this mode is turned on, the file-checking implementation is disabled, but the user should still ensure the parameters are set such that the correct file is read, otherwise an error will occur.

5 Documentation and testing

5.1 User guide


Quantics Analysis GUI
File | wsl.localhost/Ubuntu/home/lsw/GitLab/quantics/doc/analyse/anal_gui/user_guide.html
Guest

> Animated plots

Some analysis options allow you to control time using the 'scrubber' at the bottom of the screen. You can also click the play button to automatically increase time in the plot. Control how fast this goes using the `Speed` button. This button also changes the framerate if you save the video as well.

For animated 1D plots, the y-axis rescales automatically over time. To prevent this, right-click, go to `Y axis` and click `Manual` to fix the y-axis limits to their current values.

Context menus

Context menus are shown when right-clicking on the text widget or the plot widget.

> Customising the plot widget

It is possible to change the some properties of the plot by right-clicking → `Plot Options`. In this sub-menu, you can change the title of the plot (`Custom title`), whether to show the legend (`Show legend`) as well as enabling the grid/changing its opacity (`Grid`).

Figure 39: Sample of the user documentation as shown in *Google Chrome*. In the centre a short video tutorial is shown.

The Python package has a `README.md` file (see listing 5) with details on the basic installation and opening of the GUI. However, for describing the usage inside the GUI, a user guide is created. For maximum usability, the user should not have to look up any documentation, but there should still be help provided if the user needs it. This user guide contain tutorials on how to perform certain tasks in the GUI, such as navigation in the plot widget, selecting coordinates using the coodinate selector (§4.10), as well as a brief explanation of some more advanced options (§4.13). The location of the user guide is in the `doc/` directory in the *Quantics* repository. There is an HTML file which can be opened in a web browser, with an accompanying stylesheet (CSS file) and a folder for media in the guide. A link to the user documentation is also present in the “Help” menu in the menubar of the GUI.

The stylesheet describes how the HTML elements (such as paragraphs, hyperlinks, titles) are presented to the user. The design is purposefully minimalistic, using a sans-serif font to ease reading on a computer screen, and setting a maximum body width so that the user does not need to move their head to see text at the edge of their screen. The CSS file also contains a presentation for GUI buttons using the HTML `<span class="button"></span>`, useful for indicating which part of the text represent items on the GUI.

There are a number of short video tutorials embedded into the web document. These videos are generally less than 20 seconds long and provide the exact instructions for performing a task, which may be more useful for visual learners. These videos were recorded using OBS, a screen recorder. To keep the file size of the videos low, the bitrate of the video was set lower than the default value (1700 Kbps), the framerate set to 30 FPS, and the captured video was sped up by 33% using the *FFmpeg* command `ffmpeg -i video.mp4 -filter:v 'setpts=0.75*PTS' -an out.mp4` (framerate is unchanged by this command). The `-an` flag removes the audio track from the video, since an audio track is still saved even if all input devices are muted in OBS.

To embed these videos in an HTML document the `<video>` tag is used. To allow autoplay of the video, which prevents the user needing to interact with the webpage, the following attributes need to be included inside the tag: `autoplay`, `muted` (as most browsers refuse to autoplay videos with sound), `loop` (to make the video repeat itself), and `playsinline` (some mobile web browsers need this since videos can “pop out” on mobile devices).

A file containing developer documentation was also set out as a task. Due to time limitations, it was never produced. The codebase already contains plenty of comments and docstrings that help future developers to understand the code, so the intended goal of this documentation was on how to edit the .ui files using Qt Designer (e.g. adding widgets to parameter containers), and design principles inside the analysis classes (e.g. what circumstances would it be suitable to override `activate` or `optionSelected`).

5.2 Unit testing

Software testing is an important aspect in most programs to ensure that the behaviour of the program aligns with the user's expectations, and to identify the situations where this is not the case. At the lowest level, this is called *unit testing* where the individual methods or functions are analysed to ensure that for a given input, the output of the method or function is correct.[55]

Due to the nature of the project, testing is usually performed manually. This is because it is generally easier to navigate the GUI on the screen than by code, and that the output of most analyses are visual (in a plot), in which a human is generally required to check that the graphics look correct. However, there are some methods that can be tested automatically using code. In particular, the methods that are focused on for testing are methods that do not rely on external programs, as we only want to check the code written for this project[m], and methods whose purpose is not to solely produce a visualisation. These exclude any method that calls a Fortran program, or consist mostly of calls to an external package such as `sqlite3`.

The methods that satisfy these conditions are `readFloats` (§4.2.2), `rdtiming` (the sort functionality; §4.8.1), `calcrate` (§4.12.1), `ddpesgeo` (§4.12.3), `writeTable` (part of §4.12.2), and `findModeLabels` (§4.10.1). However, the incomplete nature of `findModeLabels` means that its implementation is not at a stage where it is particularly suitable to be tested. With the exception of `ddpesgeo`, whose unit test was not implemented due to time limitations, the aforementioned methods are tested in the `tests/` subdirectory of the `analysis_gui` package.

In order to perform unit tests the `unittest` module is used from the Python standard library. It provides a `TestCase` class which can be inherited from to obtain methods such as `assertEqual`, `assertTrue`, etc. to check the values of a certain variable is as expected. However, to begin unit testing we must open the GUI inside of a test, without making it visible. To do this we use the `TestCase` class' `setUp` and `tearDown` methods, which can be overridden to perform an action before each test (open the GUI), and after each test respectively (close the GUI). This is implemented in listing 23

```python
class TestClass(unittest.TestCase):
    def setUp(self):
        self.app = main_window.QtWidgets.QApplication(sys.argv)
        self.window = main_window.AnalysisMain()
    # test methods go here ...
    def tearDown(self):
        self.window.close()
        self.app.quit()
        del self.app # required to prevent errors with auto garbage collection
```

Listing 23: Opening a GUI inside a class such that it can be tested.

Tests for the analysis methods are stored in a file named `test_analyses.py` and the rest (`readFloats` and `writeTable`) are stored in a file named `test_misc.py`. Normally, the tests are executed using `unittest.main()` in the Python file's main method, but as the files are stored in a package, executing Python files directly doesn't work (§3.4). Instead, we can use a `unittest.TestLoader()` to execute tests outside of the package directory, using the `top_level_dir` parameter (listing 24). This will also run all files in the directory starting with `test_`, without needing to execute each file in the folder. A function which performs this is located in the `run_tests.py` file. Alternatively, one can pass the `-test` flag to the `quantics_analysis_gui.py` file that is located outside of the package that calls this function.

```python
def runTests():
    loader = unittest.TestLoader()
    tests = loader.discover('analysis_gui.tests', pattern='test_*.py',
                            top_level_dir=Path(__file__).parents[2])
    testRunner = unittest.runner.TextTestRunner(verbosity=2)
    testRunner.run(tests)
```

Listing 24: Code snippet of running test files in the `analysis_gui/tests` folder. This function can be called directly within the package.

Tests within each class are implemented using a method whose name starting with `test`, such as `testReadFloats`. For this method, a NumPy array is generated with random numbers, where the number

[m]It is still possible to test these methods using a *mock* to replace the interface to the external program. This is called *mock testing*, but is out of scope for this project.

of rows and number of columns are class constants and can be set by the developer. To ensure the function can read numbers in normal decimal format and scientific format, the mantissa and exponent are generated seperately, which provide a range of normal and very small/large numbers that can only be expressed in scientific notation. This array is then saved to a file, which `readFloat` reads. The test then asserts that the returned array is the same as the generated array. In addition, the performance can be tested using `time.perf_counter()`; a test of reading a 20000×5 grid took 0.09 s on my machine, which is approximately the same rate as mentioned in §4.2.2.

The NumPy array generated can also be used to test `writeTables`. Since this test succeeds the `readFloats` test, we know that `readFloats` works. If input the NumPy array into the text widget via `writeTables`, then `readFloats` should also be able to parse this as well. However, as `writeTables` only writes to a limited precision, `numpy.all_close` is used instead of `numpy.array_equal`.

In order to test the analysis functions we must have some example files for the analysis methods to read. These are called *test fixtures*, and are located in the `fixtures/` subdirectory, which at the moment only contains a `timing` and `log` file for the two analyses we are about to test. We must then navigate to this directory and set all parameters directly using PyQt's slot methods such as `dir.edit.setText` and `analint.analyse.click()`.

To test that the `rdtiming` method sorts correctly, a list of strings containing the correct order is given, and is converted into a regular expression by inserting `'.*?'` between each list entry. With the `re.DOTALL` flag, this allows the `'.*?'` regex to match any character between two entries as long as they are between consecutive entries. Thus, in order for there to be a match, all entries in the list must come in order. This test is performed for the first three columns, which are subroutine name, number of calls, and CPU/call. If they all succeed, it can be assumed the sort will work for all columns.

To test that the `calcrate` method sorts correctly, a list of the expected y values are given. After the analysis is performed, the actual y values can be obtained from the plot widget via `plot.listDataItems()[0].getData()[1]`, where the `[0]` selects the first line in the plot (for `calcrate` there is only one line), and the `[1]` selects the y data. The expected and obtained data must be the same for the test to succeed.

5.3 Usability testing

To improve the usability (§3.2) of a product, the software can be tested by having participants perform tasks using the software, and have feedback analysed from them. This is called *usability testing*.[56] In more formal circumstances, this may involve recording their body movements when using the software, or asking them questions as they use it. However, due to the nature of the project, the "absolute" usability is less important than the aim of the GUI which is to increase usability comparatively against the CLI. Thus we want to obtain feedback from people who work on the *Quantics* package.

Since these people in question are usually quite busy, the method used for usability testing for the project is a questionnaire, which is intended to be as quick as possible to fill out. The questionnaire was emailed to participants in the *Quantics* group by my supervisor. It is composed of two parts. Firstly, usability is quantified using a series of questions on a *Likert scale*[57], where the response to the question is a number on a scale of 1–5, where 1 is "Strongly disagree" and 5 is "Strongly agree". Questions for this survey are based on the *System usability scale* (SUS),[58] adapted with more specific questions concerning the GUI. For odd-numbered questions, 1 indicates better usability, and for even-numbered questions, 5 indicates better usability.

1. I struggled to install/open the GUI program on my computer.
2. I prefer using the GUI over the command line interface.
3. I found the GUI too complex.
4. I learnt how to use the GUI very quickly.
5. I found the GUI exhausting to use properly.
6. I understood the documentation easily.
7. I think most people would need to read the user guide before using the GUI.
8. I felt the GUI covered all the essential analysis programs in Quantics.

Some questions such as item 3, 5, and 7 are general questions the measure the principles of usability (here, satisfaction, efficiency, and effectiveness respectively) while the others relate to more specific questions about *Quantics*, such as item 2 and 8, which help gauge the relative usability compared to the CLI. Items 1 and 6 are primarily used to assess the quality of the documentation, and item 4 assesses the learning curve, which relates the the effectiveness principle.

Secondly, the participant is given asked for any additional thoughts on the GUI, for which they can write down anything they want. This is to gain feedback on specific parts of the GUI for which the questions above did not cover, such as whether there was a part of a GUI that was hard to work with, or suggestions that could improve the GUI. The program's usability can then be directly improved by implementing the code relating to the participant's feedback.

Unfortunately, due to the relatively low candidate pool of participants and that the questionnaire was sent particularly late in the project time period, not enough replies were made such that an analysis of the usability can be performed in this report.

6 *Conclusion*

6.1 Summary

The analysis stage in a quantum dynamics simulation (Fig. 1) can be considered the most important, as in this stage, the useful results and properties of interest can be extracted from the outputs of the simulation, which give interpretations to results of laboratory experiments. This analysis is performed by *Quantics*' analysis suite, of which the `analysis` command line program could be considered the most user-friendly program to perform analyses with. It involved calling other Fortran programs, typing in numbers to select options (listing 3), and used *gnuplot* to perform plots. Discussed in §1.3, the aim of the project was to go one step further, i.e. to increase usability, replacing typing on a command-line to interacting with widgets in a GUI, and replacing a *gnuplot* popup window with an embedded, native plot widget using pyqtgraph.

In order to implement this, a framework for integrating analyses was created, to core of which is the `AnalysisTab` class (§4.2). Analyses, which are repesented by class methods, are divided into categories, which are represented by a class that inherited from `AnalysisTab`. All programs called in `analysis` were re-implemented in the analysis methods except for any non-functional programs (namely `norm` and `rdsteps`). In addition to interfacing with all the Fortran programs in Python, other features from the CLI program such as options to customise the plot (§4.4) and selecting coordinates to plot a multidimensional array (§4.10) were included.

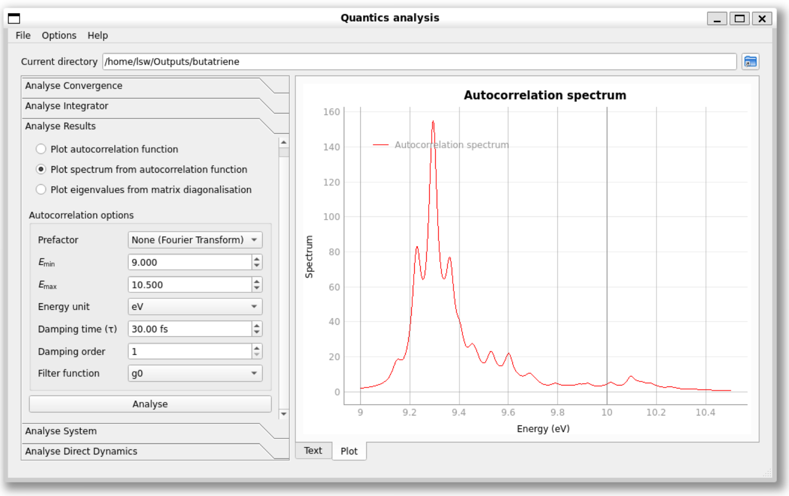
A user guide (§5.1) was created to help users install Python and the prerequisite packages, as well as to explain some more advanced features of the GUI. To help improve the code in the future, particular analyses were tested using Python's `unittest` module to ensure correct outputs, and a usability test was created such that users could give feedback on the user-friendliness of the GUI.

To show that the GUI can be used as a suitable, user-friendly replacement of the `analysis` CLI program, the analysis of the butatriene spectrum in §1.2 is re-performed using the GUI.

6.2 The butatriene cation, revisited

The only use of the CLI that is needed in this section is to open the Python file which loads the `analysis_gui` package and runs `gui.runGui()`. In particular, for the currently implemented *Quantics* directory structure, the command to open the GUI is `python3 $QUANTICS_DIR/source/analyse/quantics_analysis_gui.py`. If this command is run in the output directory (assuming the output files have already been created using the `quantics` command) the directory does not need to be changed; otherwise, it can be changed by clicking the button inside the directory widget or with `Ctrl + O`.

To plot the absorption spectrum of butatriene, we click on "Analyse Results" and click on the "Plot spectrum from autocorrelation function" radio box. A list of parameters should show, which we can input E_{min} as 9.0, E_{max} as 10.5, energy unit as eV, damping time as 30 fs, and damping order of 1. All of these values are the same as before. For the "Filter function" combo box, the default of `g0` is fine. Then, we click "Analyse". This will result in a screen per Fig. 40.


Current directory /home/lsw/Outputs/butatriene

Analyse Results

( ) Plot autocorrelation function
(●) Plot spectrum from autocorrelation function
( ) Plot eigenvalues from matrix diagonalisation

Autocorrelation options

Prefactor	None (Fourier Transform)
E_{min}	9.000
E_{max}	10.500
Energy unit	eV
Damping time (τ)	30.00 fs
Damping order	1
Filter function	g0

Figure 40: Absorption spectrum from the excitation to the lower state S_1 as seen in the GUI.

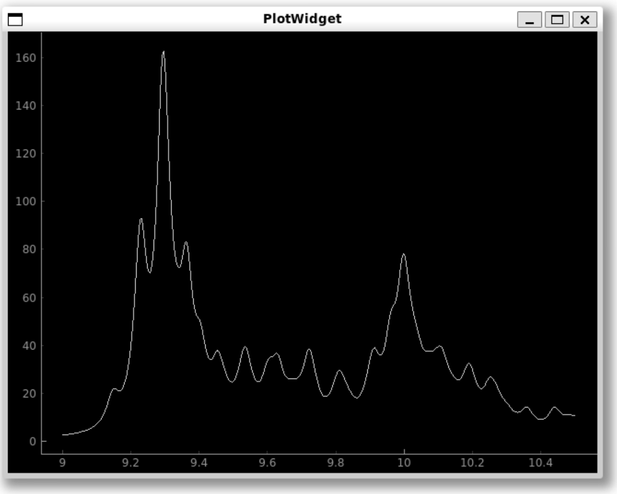


Figure 41: Combined absorption spectrum using the code in listing 25

```python
import numpy as np
import pyqtgraph as pg
s1 = np.load('s1.npy')
s2 = np.load('s2.npy')
x = s1[:, 0]
y = s1[:, 1] + s2[:, 1]
pg.plot(x, y)
pg.exec()
```

Listing 25: Summation of spectra using NumPy arrays, then plotted using pyqtgraph.

Unfortunately, the combination of spectra cannot be performed inside the GUI as of yet. However, it is still possible to use the GUI to save the underlying NumPy arrays, which can then be combined. To do this, we right click on the graph and click "Save .npy data", then save this file with name `s1.npy` (for example) in a known directory. Navigate to the other output directory—all parameters have already been set so simply click "Analyse" again. Save this file with name `s2.npy` inside the same directory as `s1.npy`. These .npy files can be loaded in a Python script, summed, and plotted using listing 25, either by writing a Python file and executing it or calling `python3` on the command line. Alternatively, *matplotlib* can be used instead of pyqtgraph for more customisable, publication-quality graphics.

Despite the limitations of the GUI in this case, hopefully it is still clear how using the GUI can be more intuitive, faster, and more convenient than the CLI. The next section, §6.3, details more of these limitations.

6.3 Limitations and future work

While a substantive amount of work went into ensuring the project is complete as possible, there are many limitations to the project. Some limitations have already been discussed in the relevant implementation sections. In general, the limitations for this project can be categorised into two areas: limitations due to lack of time, and limitations due to the Fortran code. Both of these categories can be fixed with future work, and as such, this section also covers recommendations for future work in the *Quantics* package.

6.3.1 Using F2Py

F2Py, which is short for Fortran to Python interface generator, is a NumPy subpackage that provides a direct connection between Fortran and Python code. In particular, Fortran arrays can be directly accessed and retrieved from memory without the need to write the file to the disk by the Fortran program, then read the file from the disk by Python. This would be essentially equivalent to the "Analysis GUI" and "Existing Fortran analysis programs" merged together in Fig. 10.

Implementing this requires knowledge of the *Quantics* Fortran codebase. *Quantics* is a very large project, and its GitLab repository already contains 225.1 MiB of data. If this task is to be implemented, it should be relegated to someone who already has prior experience working with Fortran and *Quantics*.

However, perhaps a more suitable question is to ask whether using *F2Py* would speed up the GUI significantly. While fetching the array from memory is definitely quicker than writing to disk then retrieving it, the implementation for reading floats in the `AnalysisTab` class can parse around one million floats per second, using calculations from §4.2.2 and §5.2, which is fast enough to be unnoticeable in almost all cases. The slowest part of the GUI implemented so far is probably the splitting of the 2D NumPy array to generate a 3D array in §4.9.1 (of which there may be a quicker implementation possible). As such, while the use of *F2Py* would be an improvement, the gain in speed is too marginal to recommend working on.

There are other potential uses of *F2Py* which are mentioned in §6.3.3.

6.3.2 Silent errors

A problem with interfacing with command line problems that had been previously mentioned in §4.5 is that the output of a command line program needs to be consistent such that the following code written in Python can succeed. Should this consistency fail the command line program should return a non-zero exit code. However, few of the *Quantics* analysis programs written in Fortran do this, which leads to errors such as in Fig. 25, where the error produced is related to a failure in the following code rather than the actual error which preceded it.

However, there is a slightly more dangerous circumstance which occurs if there is no error raised by the following code. The situation in Fig. 25 happened because a blank file was produced by the analysis program which `readFloats` failed to read. Some programs return exit code 0 on failure and do not produce any file, namely `rdgpop`. For this program, as the output's filename (`gpop.pl`) is fixed, if a `gpop.pl` already exists in the directory and `rdgpop` returns an error, the GUI will plot the file that already exists in the directory rather than show an error, which is extremely misleading. Such occurences are named *silent errors*—no error popup is produced and the only way the user would know that there was an error is through inspecting the command line output in the text widget.

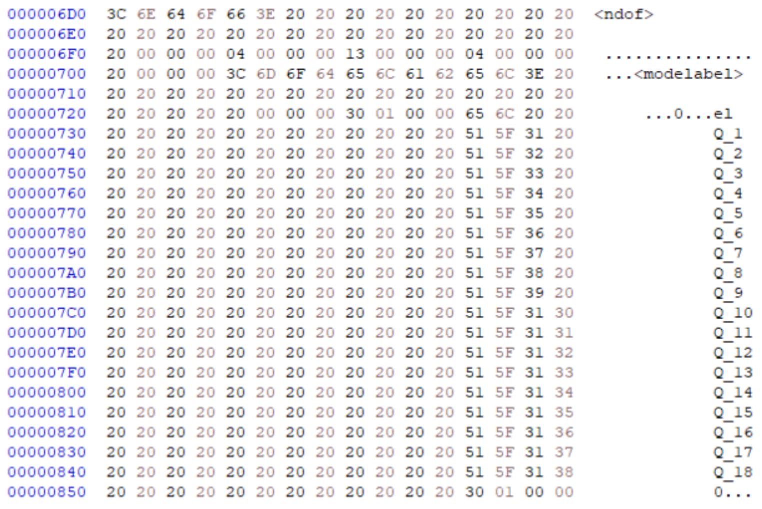
While there are Python implementations to fix this issue (such as checking whether a file exists beforehand, or deleting the output file after the analysis is complete), the problem can be resolved by simply ensuring a non-zero exit code is returned if there is an error with the analysis program. If the Fortran program is terminated using STOP, then simply writing a number afterwards will generate the exit code, e.g. STOP 1 will terminate the program with exit code 1.

6.3.3 Input file parser

While it is certainly good to catch and raise any errors and explain them to the user, it is better to prevent errors happening at all. In particular, errors that occur due to invalid parameters, such as selecting a state higher than the total number of states, or a mode index higher than the total number of modes, can be prevented if the maximum value of the spinbox is set to the number of states/modes. This can be done by overriding `optionSelected` in the analysis class and using something like `self.spinbox.setMaximum(n_state)`. However, the number of states/modes needs to be known for this to happen.

This information can be obtained from the `input` file. This has already been used for the number of GWPs in `gwptraj` (§4.7.4) and the mode labels in §4.10. However, there are hundreds of possible parameters that can be used in the `input` file, and the implementation used in §4.10.1 only covers a limited range of possible cases. For the rest of the cases the user has to manually input these values, which is not particularly desirable.

An input file parser would be useful to solve these problems. It can be a class instance belonging to the main window, which is refreshed when the directory is changed by the user. Useful information can then be extracted from this instance when necessary. Technically, a parser already exists in the *Quantics* package, named `mctdhinp.py`. However, it is extremely outdated (originally written in Python 2), can only extract a limited number of parameter values, and has unintuitive methods and attributes. As such it was decided the existing parser was not suitable for the GUI and manual methods to read the input file were implemented as mentioned above.



```
000006D0  3C 6E 64 6F 66 3E 20 20 20 20 20 20 20 20 20 20  <ndof>
000006E0  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
000006F0  20 00 00 00 04 00 00 00 13 00 00 00 04 00 00 00   ................
00000700  20 00 00 00 3C 6D 6F 64 65 6C 61 62 65 6C 3E 20   ...<modelabel>
00000710  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
00000720  20 20 20 20 20 00 00 00 30 01 00 00 65 6C 20 20        ...0...el
00000730  20 20 20 20 20 20 20 20 20 20 20 20 51 5F 31 20               Q_1
00000740  20 20 20 20 20 20 20 20 20 20 20 20 51 5F 32 20               Q_2
00000750  20 20 20 20 20 20 20 20 20 20 20 20 51 5F 33 20               Q_3
00000760  20 20 20 20 20 20 20 20 20 20 20 20 51 5F 34 20               Q_4
00000770  20 20 20 20 20 20 20 20 20 20 20 20 51 5F 35 20               Q_5
00000780  20 20 20 20 20 20 20 20 20 20 20 20 51 5F 36 20               Q_6
00000790  20 20 20 20 20 20 20 20 20 20 20 20 51 5F 37 20               Q_7
000007A0  20 20 20 20 20 20 20 20 20 20 20 20 51 5F 38 20               Q_8
000007B0  20 20 20 20 20 20 20 20 20 20 20 20 51 5F 39 20               Q_9
000007C0  20 20 20 20 20 20 20 20 20 20 20 20 51 5F 31 30               Q_10
000007D0  20 20 20 20 20 20 20 20 20 20 20 20 51 5F 31 31               Q_11
000007E0  20 20 20 20 20 20 20 20 20 20 20 20 51 5F 31 32               Q_12
000007F0  20 20 20 20 20 20 20 20 20 20 20 20 51 5F 31 33               Q_13
00000800  20 20 20 20 20 20 20 20 20 20 20 20 51 5F 31 34               Q_14
00000810  20 20 20 20 20 20 20 20 20 20 20 20 51 5F 31 35               Q_15
00000820  20 20 20 20 20 20 20 20 20 20 20 20 51 5F 31 36               Q_16
00000830  20 20 20 20 20 20 20 20 20 20 20 20 51 5F 31 37               Q_17
00000840  20 20 20 20 20 20 20 20 20 20 20 20 51 5F 31 38               Q_18
00000850  20 20 20 20 20 20 20 20 20 20 20 20 30 01 00 00               0...
```

Figure 42: The number of DOFs and mode labels shown in the `dvr` header using a hex editor.

Alternatively, some relevant information are stored in the "headers" of some files such as `psi` and `dvr` (Fig. 42). For Fortran programs that require these files, this is the implementation that is used to extract information such as a list of mode labels without needing to reading the input file. However, these headers are written in a binary format which is not particularly easy to extract information from. If the relevant subroutine which reads these header can be wrapped in a Python function (possibly using F2Py), this could potentially save the enormous effort of writing code that can parse the input file.

6.3.4 `CoordinateSelector` and menu-based programs

The `CoordinateSelector` widget as discussed in §4.10 writes the correct input for option 20 (Listing 2, coordinate selection) in the menu-based program `showsys`. However, it should also be able to change the axis limits/bounds of x and y, and provide and change the units of the DOFs. In the `showsys` menu, these options are located seperately as option 30 and 60/90, respectively. Fustratingly, changing the values for these options require the *mode index* rather than the *mode label*, which require the DOF to be specified as the index of the order in which the DOF is given in the `input` file, similarly to the parameter in `showd1d` (§4.9.1).

To implement this in Python would require much more complicated code to parse the input file (also discussed in the section above). It would be potentially easier to simply combine the options together in the Fortran program, allowing specifications such as `modelabel1 x -4.0 4.0 au` and `modelabel2 0.1 au`. Even better would to have this implemented as an alternate parameter-type interface such that navigating the menus using code is not necessary at all.

6.3.5 3D plots

While pyqtgraph has support for 3D plots, they require an external package known as `pyopengl` which provides Python bindings to OpenGL. I could not get this working on my machine, so any 3D plots did not work and thus were not implemented. This is unfortunate as a surface plot is a particularly useful visualisation to view potential energy surfaces, such as in Fig. 8.

In addition, the pyqtgraph API for 3D plots is still relatively undeveloped (e.g. the use of `pyqtgraph.exporters` does not function for 3D plots), and is subject to change as the pyqtgraph developers are considering using a package known as *VisPy* to provide alternative OpenGL bindings[n]. Alternatively, if

[n]https://pyqtgraph.readthedocs.io/en/pyqtgraph-0.13.3/api_reference/3dgraphics/index.html

the use of 3D is of the utmost importance, the more mature *matplotlib* package can produce a static image of a 3D plot which can be embedded in the plot widget.

6.3.6 Other analysis programs

As mentioned in §6.1, all functioning analyses included in the `analysis` program were implemented in the GUI. However, there are still many other useful programs included in *Quantics* that were not included in the GUI such as `showspf`, which plots the SPFs, and `adpop`, which plots the adiabatic state populations. There are far too many to be feasibly included in the timeframe of the project.

There are also programs such as `rddddb` and `ddtraj`, used for DD-vMCG simulations, that were considered as possible candidates for integration. In particular, `rddddb` produces a geometry for each entry, and `ddtraj` shows the molecular structure along a trajectory. These programs produced a `plot.db` file containing data in a *Gaussian*-type format that could be read by a molecular structure viewer such as *Jmol*.

However, just as we did not want to plot to open in a new window such as in *gnuplot*, instead opting to plot graphs natively inside the GUI using pyqtgraph, the same would ideally go for *Jmol*. However, there is no equivalent, suitable Python package that can be integrated easily in a PyQt window. A possible solution is to use the JavaScript package *JSmol* and embed this in the GUI using a web browser, using a seperate package named *PyQtWebEngine*. A local server can be run serving a basic webpage interfacing with *JSmol*.

However, *JSmol* is not as efficient as *Jmol* as it mostly contains code auto-translated from Java. The `plot.db` files produced by `rddddb` and `ddtraj` methods typically contain a list of thousands of geometries which generate a large amount of latency in *JSmol*. As such, the best solution at the moment is to simply open a *Jmol* window, or some molecular geometry viewer that uses OpenGL is found which can be embedded in PyQt's `QOpenGLWidget`.

6.3.7 Miscellaneous

As with all things written in code, there may be bugs in the GUI, where the functionality does not work as expected or gives an error where there should not be. While great effort has been used to fix these bugs, there are likely many more which are not yet found, especially relating to the interface with menu-based programs, which is rather tentative. A user can report any bugs to the GitLab issues page, detailing the problematic files and the steps to reproduce them. However, as a limited number of people work on the *Quantics* package this feature is rarely used.

If the keywords `time-not-fs` and `energy-not-ev` are specified in the `input` file the times and energies are given in atomic units. In this case a flag such as `-nofs` must then be given using the additional options box to correct the values. However, the units given in the labels are fixed strings in the code, and this will end up producing the wrong unit in the plot. This could be fixed by manually adding a `nofs` checkbox in the relevant parameter container, at the cost of complicating the code a little.

Analyses that read the QC database `database.sql` must currently navigate to the folder that the file is in, as typically this folder is in a different directory than the output files. It would be useful if the GUI can read the database from the output file folder without needing to navigate to a seperate directory. This can be done by reading the relevant part of the `input` file, then implementing an attribute with a getter and setter in the analysis class that refreshes when the user enters a directory. The `required_files` dictionary can then contain this attribute instead of the default `database.sql`.

Finally, the project contained a case of "re-inventing the wheel". Namely, the `readFloats` function (§4.2.2) performs an identical goal to the NumPy function `numpy.loadtxt` (or `numpy.genfromtxt` which is more customisable as it has more parameters), which was only "discovered" after the method had already been implemented. While the existing implementation is already very fast, one may consider replacing the current implementation with the more efficient NumPy method. Additionally, there are PyQt modules that can interface with SQL databases, namely `QtSql.QSqlDatabase`. This could potentially replace something like `querydb` (§4.12.2) but would require more work to integrate this widget into the .ui file.

7 Bibliography

[1] H.-D. Meyer and G. A. Worth. Quantum molecular dynamics: propagating wavepackets and density operators using the multiconfiguration time-dependent Hartree method. *Theoretical Chemistry Accounts*, 109(5):251–267, 2003. doi:10.1007/s00214-003-0439-1.

[2] A. H. Zewail. Laser Femtochemistry. *Science*, 242(4886):1645–1653, 1988. doi:10.1126/science.242.4886.1645.

[3] G. A. Worth, H.-D. Meyer, K. Giri, G. W. Richings, M. H. Beck, A. Jäckle, et al. *The Quantics Package.* University College London, London, 2023. URL `https://gitlab.com/quantics/quantics/`.

[4] H.-D. Meyer, U. Manthe, and L. S. Cederbaum. The multi-configurational time-dependent Hartree approach. *Chemical Physics Letters*, 165(1):73–78, 1990. doi:10.1016/0009-2614(90)87014-I.

[5] G. A. Worth. Quantics: A general purpose package for quantum molecular dynamics simulations. *Computer Physics Communications*, 248:107040, 2020. doi:10.1016/j.cpc.2019.107040.

[6] M. H. Beck, A. Jäckle, G. A. Worth, and H.-D. Meyer. The multiconfiguration time-dependent Hartree (MCTDH) method: a highly efficient algorithm for propagating wavepackets. *Physics Reports*, 324(1): 1–105, 2000. doi:10.1016/S0370-1573(99)00047-2.

[7] Chr. Cattarius, G. A. Worth, H.-D. Meyer, and L. S. Cederbaum. All mode dynamics at the conical intersection of an octa-atomic molecule: Multi-configuration time-dependent Hartree (MCTDH) investigation on the butatriene cation. *The Journal of Chemical Physics*, 115(5):2088–2100, 2001. doi:10.1063/1.1384872.

[8] F. Brogli, E. Heilbronner, E. Kloster-Jensen, A. Schmelzer, A. S. Manocha, J. A. Pople, and L. Radom. The photoelectron spectrum of butatriene. *Chemical Physics*, 4(1):107–119, 1974. doi:10.1016/0301-0104(74)80051-0.

[9] H.-D. Meyer. Studying molecular quantum dynamics with the multiconfiguration time-dependent Hartree method. *WIREs Computational Molecular Science*, 2(2):351–374, 2012. doi:10.1002/wcms.87.

[10] L.S. Cederbaum, W. Domcke, H. Köppel, and W. Von Niessen. Strong vibronic coupling effects in ionization spectra: The "mystery band" of butatriene. *Chemical Physics*, 26(2):169–177, 1977. ISSN 0301-0104. doi:10.1016/0301-0104(77)87041-9.

[11] A. D. N. Edwards. The rise of the graphical user interface. *Information Technology and Disabilities E-Journal*, 2(4):3, 1995. URL `http://itd.athenpro.org/volume2/number4/article3.html`.

[12] M. Fitzpatrick. *Create GUI Applications with Python & Qt5.* Independently published, 5th edition, 2022. ISBN 9798831846126. URL `https://www.pythonguis.com/pyqt5-book/`.

[13] S. Chacon and B. Straub. *Pro Git.* Apress, New York, 2nd edition, 2014. ISBN 9781484200773. URL `https://git-scm.com/book/en/v2`.

[14] D. A. McQuarrie and J. D. Simon. *Physical chemistry: A molecular approach.* University Science Books, New York, 1997. ISBN 9780935702996. URL `https://uscibooks.aip.org/books/physical-chemistry-a-molecular-approach/`.

[15] E. J. Heller. Time-dependent approach to semiclassical dynamics. *The Journal of Chemical Physics*, 62 (4):1544–1555, 1975. doi:10.1063/1.430620.

[16] H.-D. Meyer, F. Gatti, and G. A. Worth. *Multidimensional Quantum Dynamics.* John Wiley & Sons, Weinheim, 2009. ISBN 9783527627400. doi:10.1002/9783527627400.

[17] A. Raab. On the Dirac–Frenkel/McLachlan variational principle. *Chemical Physics Letters*, 319(5):674–678, 2000. doi:10.1016/S0009-2614(00)00200-1.

[18] F. Gatti, B. Lasorne, H.-D. Meyer, and A. Nauts. *Applications of Quantum Dynamics in Chemistry*, volume 98 of *Lecture Notes in Chemistry.* Springer, Cham, 2017. ISBN 9783319539218. doi:10.1007/978-3-319-53923-2.

[19] A. D. McLachlan. A variational solution of the time-dependent Schrödinger equation. *Molecular Physics*, 8(1):39–44, 1964. doi:10.1080/00268976400100041.

[20] R. B. Gerber, M. A. Ratner, and V. Buch. Simplified time-dependent self-consistent field approximation for intramolecular dynamics. *Chemical Physics Letters*, 91(3):173–177, 1982. doi:10.1016/0009-2614(82)83635-X.

[21] P. Jungwirth and R. B. Gerber. Quantum dynamics of large polyatomic systems using a classically based separable potential method. *The Journal of Chemical Physics*, 102(15):6046–6056, 1995. doi:10.1063/1.469339.

[22] Z. Kotler, A. Nitzan, and R. Kosloff. Multiconfiguration time-dependent self-consistent field approximation for curve crossing in presence of a bath. A fast fourier transform study. *Chemical Physics Letters*, 153(6): 483–489, 1988. doi:10.1016/0009-2614(88)85247-3.

[23] A. P. J. Jansen. A multiconfiguration time-dependent Hartree approximation based on natural single-particle states. *The Journal of chemical physics*, 99(5):4055–4063, 1993. doi:10.1063/1.466101.

[24] M. H. Beck and H.-D. Meyer. An efficient and robust integration scheme for the equations of motion of the multiconfiguration time-dependent Hartree (MCTDH) method. *Zeitschrift für Physik D Atoms, Molecules and Clusters*, 42:113–129, 1997. doi:10.1007/s004600050342.

[25] U. Manthe, H.-D. Meyer, and L. S. Cederbaum. Wave-packet dynamics within the multiconfiguration Hartree framework: General aspects and application to NOCl. *The Journal of chemical physics*, 97(5): 3199–3213, 1992. doi:10.1063/1.463007.

[26] U. Manthe, H.-D. Meyer, and L. S. Cederbaum. Multiconfigurational time-dependent Hartree study of complex dynamics: Photodissociation of NO_2. *The Journal of chemical physics*, 97(12):9062–9071, 1992. doi:10.1063/1.463332.

[27] A. Capellini and A. P. J. Jansen. Convergence study of multi-configuration time-dependent Hartree simulations: H_2 scattering from LiF (001). *The Journal of chemical physics*, 104(9):3366–3372, 1996. doi:10.1063/1.471040.

[28] A. Jäckle and H.-D. Meyer. Reactive scattering using the multiconfiguration time-dependent Hartree approximation: General aspects and application to the collinear $H + H_2 \longrightarrow H_2 + H$ reaction. *The Journal of chemical physics*, 102(14):5605–5615, 1995. doi:10.1063/1.469292.

[29] G. A. Worth, H.-D. Meyer, and L. S. Cederbaum. The effect of a model environment on the S_2 absorption spectrum of pyrazine: A wave packet study treating all 24 vibrational modes. *The Journal of chemical physics*, 105(11):4412–4426, 1996. doi:10.1063/1.472327.

[30] H. Wang and M. Thoss. Multilayer formulation of the multiconfiguration time-dependent Hartree theory. *The Journal of Chemical Physics*, 119(3):1289–1299, 2003. doi:10.1063/1.1580111.

[31] O. Vendrell and H.-D. Meyer. Multilayer multiconfiguration time-dependent Hartree method: Implementation and applications to a Henon–Heiles Hamiltonian and to pyrazine. *The Journal of Chemical Physics*, 134(4):044135, 2011. doi:10.1063/1.3535541.

[32] U. Manthe. A multilayer multiconfigurational time-dependent Hartree approach for quantum dynamics on general potential energy surfaces. *The Journal of Chemical Physics*, 128(16):164116, 2008. doi:10.1063/1.2902982.

[33] H. Wang and M. Thoss. From coherent motion to localization: dynamics of the spin-boson model at zero temperature. *New Journal of Physics*, 10(11):115005, 2008. doi:10.1088/1367-2630/10/11/115005.

[34] H. Wang and M. Thoss. From coherent motion to localization: II. Dynamics of the spin-boson model with sub-Ohmic spectral density at zero temperature. *Chemical Physics*, 370(1):78–86, 2010. doi:10.1016/j.chemphys.2010.02.027.

[35] R. Borrelli, M. Thoss, H. Wang, and W. Domcke. Quantum dynamics of electron-transfer reactions: photoinduced intermolecular electron transfer in a porphyrin–quinone complex. *Molecular Physics*, 110 (9–10):751–763, 2012. doi:10.1080/00268976.2012.676211.

[36] I. Kondov, M. Čížek, C. Benesch, H. Wang, and M. Thoss. Quantum Dynamics of Photoinduced Electron-Transfer Reactions in Dye−Semiconductor Systems: First-Principles Description and Application to Coumarin 343−TiO_2. *The Journal of Physical Chemistry C*, 111(32):11970–11981, 2007. doi:10.1021/jp072217m.

[37] G. Christopoulou, A. Freibert, and G. A. Worth. Improved algorithm for the direct dynamics variational multi-configurational Gaussian method. *The Journal of Chemical Physics*, 154(12):124127, 2021. doi:10.1063/5.0043720.

[38] G. A. Worth, M. A. Robb, and I. Burghardt. A novel algorithm for non-adiabatic direct dynamics using variational Gaussian wavepackets. *Faraday Discussions*, 127:307–323, 2004. doi:10.1039/B314253A.

[39] I. Burghardt, H.-D. Meyer, and L. S. Cederbaum. Approaches to the approximate treatment of complex molecular systems by the multiconfiguration time-dependent Hartree method. *The Journal of Chemical Physics*, 111(7):2927–2939, 1999. doi:10.1063/1.479574.

[40] G. W. Richings, I. Polyak, K. E. Spinlove, G. A. Worth, I. Burghardt, and B. Lasorne. Quantum dynamics simulations using Gaussian wavepackets: the vMCG method. *International Reviews in Physical Chemistry*, 34(2):269–308, 2015. doi:10.1080/0144235X.2015.1051354.

[41] G. W. Richings and G. A. Worth. A Practical Diabatisation Scheme for Use with the Direct-Dynamics Variational Multi-Configuration Gaussian Method. *The Journal of Physical Chemistry A*, 119(50):12457–12470, 2015. doi:10.1021/acs.jpca.5b07921.

[42] M. Baer. *Beyond Born-Oppenheimer: Electronic Nonadiabatic Coupling Terms and Conical Intersections*. John Wiley & Sons, Hoboken, 2006. ISBN 9780471780076. doi:10.1002/0471780081.

[43] G. W. Richings and G. A. Worth. Multi-state non-adiabatic direct-dynamics on propagated diabatic potential energy surfaces. *Chemical Physics Letters*, 683:606–612, 2017. doi:10.1016/j.cplett.2017.03.032.

[44] I. W. Cotton and F. S. Greatorex. Data Structures and Techniques for Remote Computer Graphics. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), page 533–544, New York, 1968. Association for Computing Machinery. ISBN 9781450378994. doi:10.1145/1476589.1476661.

[45] D. Stone, C. Jarrett, M. Woodroffe, and S. Minocha. *User Interface Design and Evaluation*. Elsevier Science, Amsterdam, 2005. ISBN 9780080520322. URL `https://oro.open.ac.uk/21270/`.

[46] ISO 9241-11:2018. *Ergonomics of human-system interaction*. International Organization for Standardization, Vernier, 2018. URL `https://www.iso.org/standard/63500.html`.

[47] J. Nielsen. Enhancing the explanatory power of usability heuristics. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '94, pages 152–158, New York, 1994. Association for Computing Machinery. ISBN 0897916506. doi:10.1145/191666.191729.

[48] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *Association for Computing Machinery SIGPLAN OOPS Messenger*, 1(1):7–87, 1990. doi:10.1145/382192.383004.

[49] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education, Harlow, Pearson new international edition, 2014. ISBN 9781292025940.

[50] G. van Rossum, B. Warsaw, and N. Coghlan. PEP8 – Style Guide for Python Code, 2013. URL `https://peps.python.org/pep-0008`.

[51] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, et al. Array programming with NumPy. *Nature*, 585:357–362, 2020. doi:10.1038/s41586-020-2649-2.

[52] B. Cannon. Idiomatic Python: EAFP versus LBYL, 2016. URL `https://devblogs.microsoft.com/python/idiomatic-python-eafp-versus-lbyl/`.

[53] V. Engel. The calculation of autocorrelation functions for spectroscopy. *Chemical Physics Letters*, 189(1): 76–78, 1992. doi:10.1016/0009-2614(92)85155-4.

[54] R. Schinke. *Photodissociation dynamics: Spectroscopy and Fragmentation of Small Polyatomic Molecules*. Cambridge University Press, Cambridge, 1993. ISBN 9780511586453. doi:10.1017/CBO9780511586453.

[55] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, Cambridge, 2nd edition, 2016. ISBN 9781107172012. doi:10.1017/9781316771273.

[56] J. S. Dumas and J. Redish. *A Practical Guide to Usability Testing*. Intellect Books, Exeter, 1999. ISBN 9781841500201. URL `https://books.google.co.uk/books?id=4lge5k_F9EwC`.

[57] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140):55, 1932. URL `https://psycnet.apa.org/record/1933-01885-001`.

[58] J. Brooke. SUS: A quick and dirty usability scale. In *Usability Evaluation in Industry*, chapter 21, pages 189–194. Taylor & Francis, London, 1996. URL `https://www.researchgate.net/publication/228593520`.

8 Appendices

8.1 Explanation of regular expressions

§4.2.2: regex to capture floating point values

`[+-]?`	optionally a + or - at the beginning
`\d+(?:\.\d*)?`	a string of digits, optionally with decimal point and possibly more digits
`(?:[eE][+-]?\d+)?`	optionally an exponential (e+N, e-N, EN) at the end
`|`	alternative: matches either the above or below
`\.\d+`	catches floats that start with decimal like .25

§4.3.2: regex to capture dates

`\w{3}`	matches three characters (the month: Jan Feb Mar, etc.)
`\s+`	matches one or more whitespace
`\d{1,2}`	matches one or two digits (the month: up to 12)
`\s+`	matches one or more whitespace
`\d{2}:\d{2}:\d{2}`	matches the time: two digits (hour), colon, two digits (minutes), colon, two digits (seconds)

§4.7.3: regex to capture `ortho` output

`#.*?`	matches the header, which starts with a # symbol and any number of characters after it up to the symbol below. the ? means to match as little as possible ("lazy") rather than the default match as much as possible ("greedy")
`\n`	matches a new line
`(.*)`	only this part is captured and returned from The regex. with the DOTALL flag this captures anything up to the symbol below
`#`	matches the first character of the header below the grid

§4.8.1: regex to split `timing` file

`(?<=Clock)\n`	the re.split splits on any match the regex makes, not only any groups captured with (), which has a different meaning in re.split (see python documentation). to make matches but not keep them we have a positive lookbehind. a new line is matched (this is deleted and splits the string) only if there is the text 'Clock' preceding it, which is the name of the last column header
`|`	alternative: matches either the above or below
`\n(?=Total)`	similarly, the positive lookahead means match a new line only if the text 'Total' succeeds it. this is the next word after the subroutine grid, as it gives the total time taken

§4.10.1: regex to extract information from SPF-SECTION

`S(?:PF-)?`	match S or SPF-
`BASIS-SECTION`	match BASIS-SECTION
`\s*\n`	match any number of whitespace after, then a new line
`(.*)`	capture any sequence of characters here until the following regex (with re.DOTALL flag)
`\nend-s(?:pf-)?`	match s or spf- after a new line
`basis-section`	match basis-section
	note this regex is also run with the re.IGNORECASE flag as the section start and end titles do not necessarily have the given case

§4.10.1: regex to find mode labels in SPF-SECTION

```
(.+?)                   match one or more characters (lazy), including commas and
                        whitespace (this is then split by comma and have
                        surrounding whitespace removed)
=                       match an equals sign
(?:                     match, but do not capture this part
   [ \d,]               match a space, digit, or comma
   |                    or
   id                   match the characters 'id' (special keyword)
)*                      any number of the last part
```

8.2 Input and operator files for butatriene

Input file

```
RUN-SECTION
name = butatriene

propagation
genoper
tfinal = 100.0   tout = 1.00
title  = Butatriene, linear coupling
auto=twice  gridpop  psi
end-run-section

OPERATOR-SECTION
opname = C4H4linear
end-operator-section

SPF-BASIS-SECTION
multi-set
     Q_5,Q_14     = 8,8
     Q_12,Q_15    = 7,6
     Q_8          = 6,6
end-spf-basis-section

PRIMITIVE-BASIS-SECTION
     el    el      2
     Q_1    HO     10   0.0     1.0    1.0
     Q_2    HO     10   0.0     1.0    1.0
     Q_3    HO     10   0.0     1.0    1.0
     Q_4    HO     10   0.0     1.0    1.0
     Q_5    HO     35   0.0     1.0    1.0
     Q_6    HO     10   0.0     1.0    1.0
     Q_7    HO     10   0.0     1.0    1.0
     Q_8    HO     15   0.0     1.0    1.0
     Q_9    HO     10   0.0     1.0    1.0
     Q_10   HO     10   0.0     1.0    1.0
     Q_11   HO     10   0.0     1.0    1.0
     Q_12   HO     10   0.0     1.0    1.0
     Q_13   HO     10   0.0     1.0    1.0
     Q_14   HO     15   0.0     1.0    1.0
     Q_15   HO     10   0.0     1.0    1.0
     Q_16   HO     10   0.0     1.0    1.0
     Q_17   HO     10   0.0     1.0    1.0
     Q_18   HO     10   0.0     1.0    1.0
end-primitive-basis-section

INTEGRATOR-SECTION
CMF/var = 0.005 , 1.0d-5
BS/spf  =  8   , 1.0d-5
```

```
SIL/A   =  10 , 1.0d-5
end-integrator-section

INIT_WF-SECTION
build
    init_state =  1
    Q_1     HO    0.0  0.0     1.0    1.0
    Q_2     HO    0.0  0.0     1.0    1.0
    Q_3     HO    0.0  0.0     1.0    1.0
    Q_4     HO    0.0  0.0     1.0    1.0
    Q_5     HO    0.0  0.0     1.0    1.0
    Q_6     HO    0.0  0.0     1.0    1.0
    Q_7     HO    0.0  0.0     1.0    1.0
    Q_8     HO    0.0  0.0     1.0    1.0
    Q_9     HO    0.0  0.0     1.0    1.0
    Q_10    HO    0.0  0.0     1.0    1.0
    Q_11    HO    0.0  0.0     1.0    1.0
    Q_12    HO    0.0  0.0     1.0    1.0
    Q_13    HO    0.0  0.0     1.0    1.0
    Q_14    HO    0.0  0.0     1.0    1.0
    Q_15    HO    0.0  0.0     1.0    1.0
    Q_16    HO    0.0  0.0     1.0    1.0
    Q_17    HO    0.0  0.0     1.0    1.0
    Q_18    HO    0.0  0.0     1.0    1.0
end-build
end-init_wf-section

end-input
```

Operator file

```
OP_DEFINE-SECTION
title
Butatriene linear model with 5 modes
see: Cattarius et al. JCP 115, 2088-2100 (2001)
end-title
end-op_define-section

PARAMETER-SECTION
# frequencies
omega_1       =   0.02500 , ev
omega_2       =   0.02640 , ev
omega_3       =   0.02960 , ev
omega_4       =   0.06060 , ev
omega_5       =   0.09120 , ev
omega_6       =   0.09970 , ev
omega_7       =   0.10000 , ev
omega_8       =   0.10890 , ev
omega_9       =   0.12580 , ev
omega_10      =   0.13020 , ev
omega_11      =   0.17740 , ev
omega_12      =   0.17730 , ev
omega_13      =   0.20540 , ev
omega_14      =   0.25780 , ev
omega_15      =   0.37130 , ev
omega_16      =   0.39860 , ev
omega_17      =   0.41190 , ev
omega_18      =   0.41190 , ev
# energies
E1            =   8.90140 , ev
E2            =   9.44550 , ev
```

```
# linear coupling constants (kappa)
kappa1_8      =   0.05310 , ev
kappa2_8      =   0.05940 , ev
kappa1_12     =   0.01150 , ev
kappa2_12     =   0.01000 , ev
kappa1_14     =   0.16280 , ev
kappa2_14     =  -0.34220 , ev
kappa1_15     =   0.04030 , ev
kappa2_15     =  -0.03210 , ev
# off-diagonal vibronic coupling constants (lambda)
lambda_5      =   0.28800 , ev
end-parameter-section

HAMILTONIAN-SECTION
modes   |  el   |  Q_1  |  Q_2  |  Q_3  |  Q_4
modes   |  Q_5  |  Q_6  |  Q_7  |  Q_8  |  Q_9
modes   |  Q_10 |  Q_11 |  Q_12 |  Q_13 |  Q_14
modes   |  Q_15 |  Q_16 |  Q_17 |  Q_18
omega_1           |2   KE
0.5*omega_1       |2   q^2
omega_2           |3   KE
0.5*omega_2       |3   q^2
omega_3           |4   KE
0.5*omega_3       |4   q^2
omega_4           |5   KE
0.5*omega_4       |5   q^2
omega_5           |6   KE
0.5*omega_5       |6   q^2
omega_6           |7   KE
0.5*omega_6       |7   q^2
omega_7           |8   KE
0.5*omega_7       |8   q^2
omega_8           |9   KE
0.5*omega_8       |9   q^2
omega_9           |10  KE
0.5*omega_9       |10  q^2
omega_10          |11  KE
0.5*omega_10      |11  q^2
omega_11          |12  KE
0.5*omega_11      |12  q^2
omega_12          |13  KE
0.5*omega_12      |13  q^2
omega_13          |14  KE
0.5*omega_13      |14  q^2
omega_14          |15  KE
0.5*omega_14      |15  q^2
omega_15          |16  KE
0.5*omega_15      |16  q^2
omega_16          |17  KE
0.5*omega_16      |17  q^2
omega_17          |18  KE
0.5*omega_17      |18  q^2
omega_18          |19  KE
0.5*omega_18      |19  q^2
E1                |1   S1&1
E2                |1   S2&2

lambda_5          |1 S1&2           |6   q

kappa1_8          |1 S1&1           |9   q
kappa2_8          |1 S2&2           |9   q
```

```
kappa1_12          |1 S1&1             |13  q
kappa2_12          |1 S2&2             |13  q
kappa1_14          |1 S1&1             |15  q
kappa2_14          |1 S2&2             |15  q
kappa1_15          |1 S1&1             |16  q
kappa2_15          |1 S2&2             |16  q
END-HAMILTONIAN-SECTION

END-OPERATOR
```