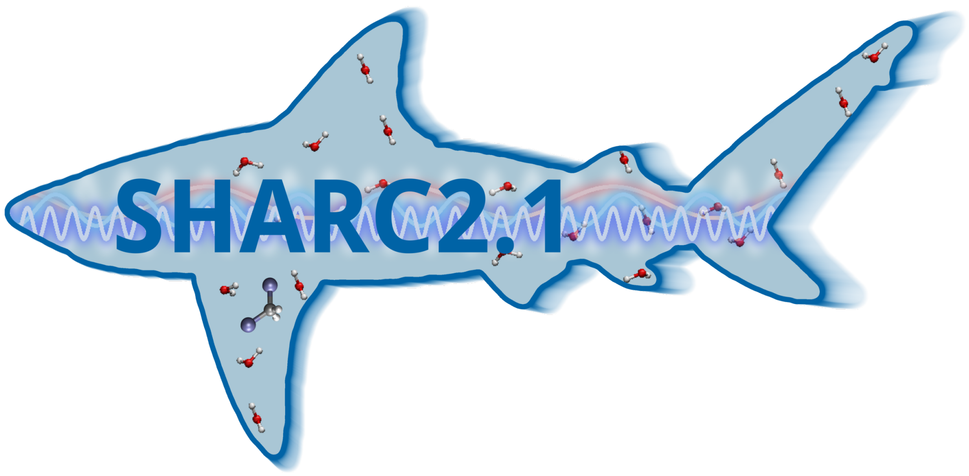

SHARC2.1: Surface Hopping Including Arbitrary Couplings

Manual

AG González
Institute of Theoretical Chemistry
University of Vienna, Austria

Vienna, 01.09.2019

Contact:

AG González
Institute of Theoretical Chemistry, University of Vienna
Währinger Straße 17
1090 Vienna, Austria

Website: sharc-md.orghttp://sharc-md.orgsharc-md.org
Email: sharc@univie.ac.atmailto:sharc@univie.ac.atsharc@univie.ac.at

Contents

1 Introduction

When a molecule is irradiated by light, a number of dynamical processes can take place, in which the molecule redistributes the energy among different electronic and vibrational degrees of freedom. Kasha's rule [**?**] states that radiationless transfer from higher excited singlet states to the lowest-lying excited singlet state (S_1) is faster than fluorescence (F). This radiationless transfer is called internal conversion (IC) and involves a changes between electronic states of the same multiplicity. If a transition occurs between electronic states of different spin, the process is called intersystem crossing (ISC). A typical ISC process is from a singlet to a triplet state, and once the lowest triplet is populated, phosphorescence (P) can take place. In figure **??**, radiative (F and P) and radiationless (IC and ISC) processes are summarized in a so-called Jabłonski diagram.

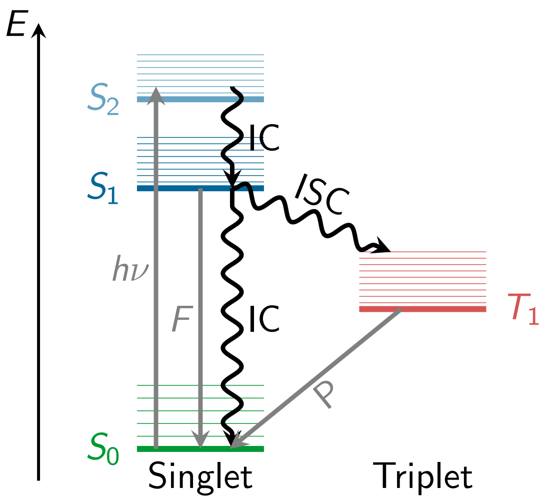

Figure 1.1: Jabłonski diagram showing the conceptual photophysical processes. Straight arrows show radiative processes: absorption ($h\nu$), fluorescence (F), and phosphorescence (P); wavy arrows show radiationless processes: internal conversion (IC) and intersystem crossing (ISC).

The non-radiative IC and ISC processes are fundamental concepts which play a decisive role in photochemistry and photobiology. IC processes are present in the excited-state dynamics of many organic and inorganic molecules, whose applications range from solar energy conversion to drug therapy. Even many, very small molecules, for example O_2 and O_3, SO_2, NO_2 and other nitrous oxides, show efficient IC, which has important consequences in atmospheric chemistry and the study of the environment and pollution. IC is also the first step of the biological process of visual perception, where the retinal moiety of rhodopsin absorbs a photon and non-radiatively performs a torsion around one of the double bonds, changing the conformation of the protein and inducing a neural signal. Similarly, protection of the human body from the influence of UV light is achieved through very efficient IC in DNA, proteins and melanins. Ultrafast IC to the electronic ground state allows quickly converting the excitation energy of the UV photons into nuclear kinetic energy, which is spread harmlessly as heat to the environment.

ISC processes are completely forbidden in the frame of the non-relativistic Schrödinger equation, but they become allowed when including spin-orbit couplings, a relativistic effect [**?**]. Spin-orbit coupling depends on the nuclear charge and becomes stronger for heavy atoms, therefore it is typically known as a "heavy atom" effect. However, it has been recently recognized that even for molecules with only first- and second-row atoms, ISC might be relevant and can be competitive in time scales with IC. A small selection of the growing number of molecules where efficient ISC in a sub-ps time scale has been predicted are SO_2 [? ? ?], benzene [?], aromatic nitrocompounds [?] or DNA nucleobases and derivatives [? ? ? ? ?].

Theoretical simulations can greatly contribute to understand non-radiative processes by following the nuclear motion on the excited-state potential energy surfaces (PES) in real time. These simulations are called excited-state dynamics simulations. Since the Born-Oppenheimer approximation is not applicable for this kind of dynamics, nonadiabatic effects need to be incorporated into the simulations.

The principal methodology to tackle excited-state dynamics simulations is to numerically integrate the time-dependent Schrödinger equation, which is usually called full quantum dynamics simulations (QD). Given accurate PESs, QD is able to match experimental accuracy. However, the need for the "a priori" knowledge of the full multi-dimensional PES renders this type of simulations quickly unfeasible for more than few degrees of freedom. Several alternative methodologies are possible to alleviate this problem. One of the most popular ones is to use surface hopping nonadiabatic dynamics.

Surface hopping was originally devised by Tully [?] and greatly improved later by the "fewest-switches criterion"[?] and it has been reviewed extensively since then, see e.g. [? ? ? ? ?]. In surface hopping, the motion of the excited-state wave packet is approximated by the motion of an ensemble of many independent, classical trajectories. Each trajectory is at every instant of time tied to one particular PES, and the nuclear motion is integrated using the gradient of this PES. However, nonadiabatic population transfer can lead to the switching of a trajectory from one PES to another PES. This switching (also called "hopping", which is the origin of the name "surface hopping") is based on a stochastic algorithm, taking into account the change of the electronic population from one time step to the next one.

The advantages of the surface hopping methodology and thus its popularity are well summarized in Ref. [?]:

- The method is conceptually simple, since it is based on classical mechanics. The nuclear propagation is based on Newton's equations and can be performed in Cartesian coordinates, avoiding any problems with curved coordinate systems as in QD.
- For the propagation of the trajectories only local information of the PESs is needed. This avoids the calculation of the full, multi-dimensional PES in advance, which is the main bottleneck of QD methods. In surface hopping dynamics, all degrees of freedom can be included in the simulation. Additionally, all necessary quantities can be calculated on-demand, usually called "on-the-fly" in this context.
- The independent trajectories can be trivially parallelized.

The strongest of these points of course is the fact that all degrees of freedom can be included easily in the calculations, allowing to describe large systems. One should note, however, that surface hopping methods in the standard formulation [? ?]—due to the classical nature of the trajectories—do not allow to treat some purely quantum-mechanical effects like tunneling, (tunneling for selected degrees of freedom is possible [?]). Additionally, quantum coherence between the electronic states is usually described poorly, because of the independent-trajectory ansatz. This can be treated with some ad-hoc corrections, e.g., in [?].

In the original surface hopping method, only nonadiabatic couplings are considered, only allowing for population transfer between electronic states of the same multiplicity (IC). The SHARC methodology is a generalization of standard surface hopping since it allows to include any type of coupling. Beyond nonadiabatic couplings (for IC), spin-orbit couplings (for ISC) or interactions of dipole moments with electric fields (to explicitly describe laser-induced processes) can be included. A number of methodologies for surface hopping including one or the other type of potential couplings have been proposed in references [? ? ? ? ? ? ?], but SHARC can include all types of potential couplings on the same footing.

The SHARC methodology is an extension to standard surface hopping which allows to include these kinds of couplings. The central idea of SHARC is to obtain a fully diagonal Hamiltonian, which is adiabatic with respect to all couplings. The diagonal Hamiltonian is obtained by unitary transformation of the Hamiltonian including all couplings. Surface hopping is conducted on the transformed electronic states. This has a number of advantages over the standard surface hopping methodology, where no diagonalization is performed:

- Potential couplings (like spin-orbit couplings and laser-dipole couplings) are usually delocalized. Surface hopping, however, rests on the assumption that the couplings are localized and hence surface hops only occur in the small region where the couplings are large. Within SHARC, by transforming away the potential couplings, additional terms of nonadiabatic (kinetic) couplings arise, which are localized.
- The potential couplings have an influence on the gradients acting on the nuclei. To a good approximation, within SHARC it is possible to include this influence in the dynamics.
- When including spin-orbit couplings for states of higher multiplicity, diagonalization solves the problem of rotational invariance of the multiplet components (see [?]).

The SHARC suite of programs is an implementation of the SHARC method. Besides the core dynamics code, it comes with a number of tools aiding in the setup, maintenance and analysis of the trajectories.

1.1 Capabilities

The main features of the SHARC suite are:

- Non-adiabatic dynamics based on the surface hopping methodology able to describe internal conversion and intersystem crossing with any number of states (singlets, doublets, triplets, or higher multiplicities).
- Algorithms for stable wave function propagation in the presence of very small or very large couplings.
- Inclusion of interactions with laser fields in the long-wavelength limit. The derivatives of the dipole moments can be included in strong-field applications.
- Propagation using either nonadiabatic couplings vectors $\langle\alpha|\frac{\partial}{\partial\mathbf{R}}|\beta\rangle$ or wave function overlaps $\langle\alpha(t_0)|\beta(t)\rangle$ (via the local diabatization procedure [?]).
- Gradients including the effects of spin-orbit couplings (with the approximation that the diabatic spin-orbit couplings are slowly varying).
- Flexible interface to quantum chemistry programs. Existing interfaces to:
  - MOLPRO 2010 and 2012: SA-CASSCF
  - OPENMOLCAS 18.0: SA-CASSCF, SS-CASPT2, MS-CASPT2, SA-CASSCF+QM/MM
  - COLUMBUS 7: SA-CASSCF, SA-RASSCF, MR-CISD
  - ADF 2017+: TD-DFT, TD-DFT+QM/MM
  - TURBOMOLE 7: ADC(2), CC2
  - GAUSSIAN 09 and 16: TD-DFT
  - ORCA 4.1: TD-DFT
  - Interface for analytical potentials
  - Interface for linear-vibronic coupling (LVC) models
- Energy-difference-based partial coupling approximation to speed up calculations [?].
- Energy-based decoherence correction [?] or augmented-FSSH decoherence correction [?].
- Calculation of Dyson norms for single-photon ionization spectra (for most interfaces) [?].
- On-the-fly wave function analysis with TheoDORE [? ? ?] (for some interfaces).
- Suite of auxiliary Python scripts for all steps of the setup procedure and for various analysis tasks.
- Comprehensive tutorial.

1.1.1 New features in SHARC Version 2.0

These features are new in SHARC Version 2.0 (2018):

- Dynamics program **`sharc.x`**:
  - New methods: AFSSH for decoherence, GFSH for hopping probabilities, reflection after frustrated hop.
  - Atom masking for size-extensive decoherence and rescaling.
  - Improved wave function and **U** matrix phase tracking.
  - Support for on-the-fly computation of Dyson norms and THEODORE descriptors.
  - Option to gracefully stop trajectories after any time step.
- Fully integrated, efficient wave function overlap program **`wfoverlap.x`**
- Quantum chemistry interfaces:
  - MOLPRO: overhauled, uses **`wfoverlap.x`**, gives consistent phase between CASSCF and CI wave functions, can do Dyson norms, parallelizes independent job parts.
  - MOLCAS: overhauled, can do (MS)-CASPT2 (only numerical gradients), QM/MM, Cholesky decomposition, Dyson norms, parallelizes independent job parts, works with OPENMOLCAS version 18.
  - COLUMBUS: overhauled, uses **`wfoverlap.x`**, can use DALTON integrals, can use MOLCAS orbitals, can do Dyson norms.

  - ANALYTICAL: —
  - TURBOMOLE: new interface, can do ADC(2) and CC2; has SOC (for ADC(2)), uses **wfoverlap.x**, works with THEODORE.
  - ADF: new interface, can do TD-DFT; has SOC, uses **wfoverlap.x**, Dyson norms, has QM/MM, works with THEODORE.
  - GAUSSIAN: new interface, can do TD-DFT; uses **wfoverlap.x**, has Dyson norms, works with THEODORE.
  - LVC: new interface, can do (analytical) linear vibronic coupling models.
- Auxilliary scripts:
  - **wigner.py**: elevated temperature sampling, LVC model setup.
  - **amber_to_initconds.py**: new script, converts AMBER trajectories to SHARC initial conditions.
  - **sharctraj_to_initconds.py**: new script, converts SHARC trajectories to SHARC initial conditions.
  - **spectrum.py**: log-normal convolution, density of state spectra.
  - **data_extractor.x**: new quantities to extract.
  - **diagnostics.py**: new script, checks all trajectories prior to analysis.
  - **populations.py**: new analysis modes.
  - **transition.py**: new script, computes total number of hops in ensemble.
  - **make_fitscript.py**: new script, prepares kinetic model fits to obtain time constants from populations.
  - **bootstrap.py**: new script, calculates error estimates for time constants.
  - **trajana_essdyn.py**: new script, performs essential dynamics analysis.
  - **trajana_nma.py**: new script, performs normal mode analysis.
  - **data_collector.py**: new script, performs generic data analysis (collecting, smoothing, convolution, integration).
  - **orca_External**: new script, allows optimization with ORCA and SHARC.
  - Several input generation helpers.
- Reworked tutorial using OPENMOLCAS (which is available at no cost).

1.1.2 New features in SHARC Version 2.1

These features are new in SHARC Version 2.1 (2019):

- Dynamics program **sharc.x**:
  - New methods: Rescaling along gradient difference vector, forced hops to ground state.
  - Output: Control of output step, writing to NetCDF format
- Dynamics program **pysharc**:
  - Dynamics driver based on Python with C extension, allows to execute the routines of **sharc.x** directly within Python, which avoids file-based interface operations. Provides extremely efficient SHARC dynamics with LVC models.
- Data extraction:
  - New options in **data_extractor.x** (e.g., populations according to [?]).
  - New extractor for NetCDF files (**data_extractor_NetCDF.x**).
  - Data converter (ASCII to NetCDF) (**data_converter.x**), useful for archiving.
- Quantum chemistry interfaces:
  - MOLCAS: now works with OPENMOLCAS version 18 or MOLCAS 8.3 and 8.4.
  - TURBOMOLE: now works with TURBOMOLE versions 7.1 to 7.4. Can now do QM/MM with TINKER.
  - ORCA: new interface, can do TD-DFT; uses **wfoverlap.x**, has Dyson norms, works with THEODORE, can do QM/MM with TINKER.
  - BAGEL: new interface, can do CASSCF, CASPT2, (X)MS-CASPT2 with analytical gradients and nonadiabatic couplings, has overlaps and Dyson norms from **wfoverlap.x**.
  - ADF interface: updated for ADF2019 and newer, not backward compatible with older ADF versions.

- Auxilliary scripts:
  - **sharcvars.sh**, **sharcvars.csh**: can be sourced to automatically set all relevant environment variables for **sharc.x**, **pysharc**, and programs using NetCDF.
  - **make_fit.py**: new script, more flexible, efficient, and integrated script for kinetic model fitting. Replaces the scripts **make_fitscript.py** and **bootstrap.py**.
  - **setup_LVCparam.py** and **create_LVCparam.py**: new setup scripts to automatically generate LVC models from quantum chemistry calculations (also without nonadiabatic coupling vectors).

1.2 References

The following references should be cited when using the SHARC suite:

- [?] .
- [?] .

Details can be found in the following references:

The theoretical background of SHARC is described in Refs. [? ? ? ? ? ? ?].

Applications of the SHARC code can be found in Refs. [?].

Other features implemented in the SHARC suite are described in the following references:

- Energy-based decoherence correction: [?].
- Augmented-FSSH decoherence correction: [?].
- Global flux SH: [?].
- Local diabatization and wave function overlap calculation: [? ? ?].
- Sampling of initial conditions from a quantum-mechanical harmonic Wigner distribution: [? ? ?].
- Excited state selection for initial condition generation: [?].
- Calculation of ring puckering parameters and their classification: [? ?].
- Normal mode analysis [? ?] and essential dynamics analysis: [? ?].
- Bootstrapping for error estimation: [?].
- Crossing point optimization: [? ?]
- Computation of ionization spectra: [? ?].
- Wave function comparison with overlaps: [?].
- Dynamics with linear vibronic coupling models: [?].
- Computation of electronic populations: [?].

The quantum chemistry programs to which interfaces with SHARC exist are described in the following sources:

- MOLPRO: [? ?],
- MOLCAS: [? ? ?],
- COLUMBUS: [? ? ? ?],
- ADF: [?],
- TURBOMOLE: [?],
- GAUSSIAN: [? ?],
- ORCA: [?],
- BAGEL: [?].

Others:

- THEODORE: [? ? ?]
- WFOVERLAP: [? ?]

1.3 Authors

The current version of the SHARC suite has been programmed by Sebastian Mai, Martin Richter, Moritz Heindl, Maximilian F. S. J. Menger, Andrew Atkins, Felix Plasser, Lea M. Ibele, Simon Kropf, and Philipp Marquetand of the AG González of the Institute of Theoretical Chemistry of the University of Vienna with contributions from Jesús González-Vázquez, Matthias Ruckenbauer, Markus Oppel, Patrick J. Zobel, and Leticia González.

1.4 Suggestions and Bug Reports

Bug reports and suggestions for possible features can be submitted to sharc@univie.ac.atmailto:sharc@univie.ac.atsharc@univie.ac.at.

1.5 Notation in this Manual

Names of programs The SHARC suite consists of Fortran90 programs as well as Python and Shell scripts. The executable Fortran90 programs are denoted by the extension **`.x`**, the Python scripts have the extension **`.py`** and the Shell scripts **`.sh`**. Within this manual, all program names are given in **`bold monospaced font`**.

Shaded Sections Important sections are given in blue boxes like the following one:

Important sections are given in blue boxes like this one.

On the other hand, examples of input files and command lines are marked like this:

```
user@host> example example.dat
```

2 Installation

2.1 How To Obtain

SHARC can be obtained from the SHARC homepage ↗ www.sharc-md.orghttp://sharc-md.orgwww.sharc-md.org. In the Download section, follow the link to GITHUB to clone or download the latest SHARC release version.

Note that you accept the Terms of Use given in the following section when you download SHARC.

2.2 Terms of Use

2.3 Installation

In order to install and run SHARC under Linux (Windows and OS X are currently not supported), you need the following:

- A Fortran90 compiler (This release is tested against GNU Fortranhttps://gcc.gnu.org/fortran/GNU Fortran 4.4.7 and Intel Fortranhttps://software.intel.com/en-us/fortran-compilersIntel Fortran 15.0).
- The BLAShttp://www.netlib.org/blas/BLAS, LAPACKhttp://www.netlib.org/lapack/LAPACK and FFTW3http://http://www.f libraries.
- Python 2https://www.python.org/downloads/release/python-2716/Python 2 (This release is tested against Python 2.6 and 2.7).
- **make**.
- [Python 3https://www.python.org/downloads/release/python-374/Python 3 if using the SHARC-ADF interface or **ADF_freq.py** (This release is tested against Python 3.5 and 3.6).]
- [Anacondahttps://www.anaconda.com/distribution/Anaconda with several libraries (**hdf5**, **hdf5_hl**, **netcdf**, **mfhdf**, **df**, **jpeg**) for the **pysharc** and NetCDF functionalities.]

Extracting The source code of the SHARC suite is distributed as a zip archive file. In order to install it, first extract the content of the archive to a suitable directory:

```
unzip sharc.zip
```

This should create a new directory called **sharc/** which contains all the necessary subdirectories and files. In figure **??** the directory structure of the complete SHARC directory is shown.

Compiling and installing (without pysharc) To compile the Fortran90 programs of the SHARC suite, go to the **source/** directory.

```
cd source/
```

and edit the **Makefile** by adjusting the variables in the top part. If you only want to install regular SHARC (no **pysharc** or NetCDF), set **USE_PYSHARC** to **false**.

Then, inside **source/** issuing the command:

```
make
```

will compile the source and create all the binaries.

```
make install
```

will additionally copy the binary files into the **sharc/bin/** directory of the SHARC distribution, which already contains all the python scripts which come with SHARC.

Compiling and installing (with pysharc) If you intend to perform computations with **pysharc** (with LVC models) or with NetCDF functionality, you need to set **USE_PYSHARC** to **true** in **source/Makefile**. Note that in this case, you also need to set a sensible path for the **ANACONDA** variable.

Due to a number of dependencies, compiling with **pysharc** is slightly more complicated than without. The simplest way to compile both **pysharc** and the regular executables together is to

1. run **make install** in **pysharc/**, then
2. run **make install** in **source/**.

Alternatively, you might want to compile the regular executables without **pysharc** or NetCDF, and then compile **pysharc**. To do this:

1. set **USE_PYSHARC** to **false**,
2. run **make install** in **source/**,
3. run **make clean** in **source/**,

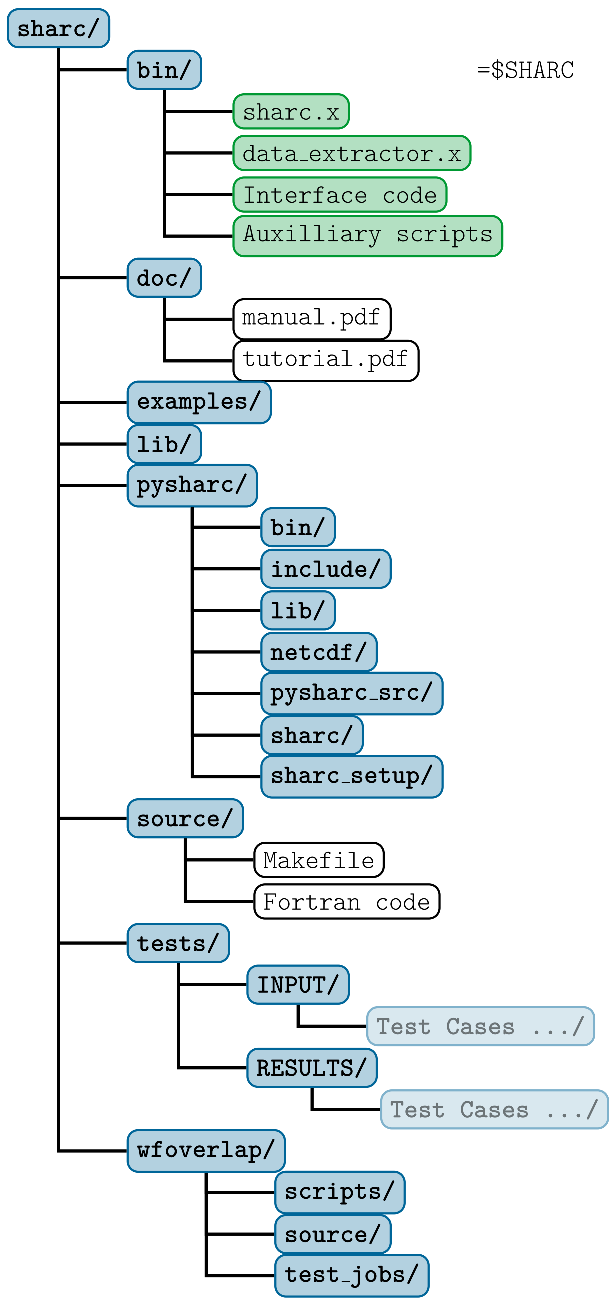

Figure 2.1: Directory tree containing a complete SHARC installation.

4. set **`USE_PYSHARC`** to **`true`**,
5. run **`make install`** in **`pysharc/`**.

Environment setup In order to use the SHARC suite, set the environment variable **`$SHARC`** to the **`bin/`** directory of the SHARC installation. This ensures that all programs of the SHARC suite find the other executables and all calls are successful. For example, if you have unpacked SHARC into your home directory, just set:

```
export SHARC=~/sharc/bin
```
(for bourne shell users)

or

```
setenv SHARC $HOME/sharc/bin   (for c-shell type users)
```

Note that it is advisable to put this line into your shell's login scripts.

More comfortably, in SHARC 2.1, you can simply source either of the **sharcvars** files (generated by **make**) that are located in the **bin/** directory:

```
source ~/sharc/bin/sharcvars.sh   (for bourne shell users)
```

or

```
source ~/sharc/bin/sharcvars.csh   (for c-shell type users)
```

2.3.1 WFOVERLAP Program

The SHARC package contains as a submodule the program WFOVERLAP, which is necessary for many functionalities of SHARC. In order to install and test this program, see section ??.

2.3.2 Libraries

SHARC requires the BLAS, LAPACK and FFTW3 libraries. During the installation, it might be necessary to alter the **LDFLAGS** string in the **Makefile**, depending on where the relevant libraries are located on your system. In this way, it is for example possible to use vendor-provided libraries like the Intel MKLhttps://software.intel.com/en-us/intel-mklIntel MKL. For more details see the **INSTALL** file which is included in the SHARC distribution.

As specified above, users of **pysharc** and the NetCDF functions will need additional libraries (**hdf5**, **hdf5_hl**, **netcdf**, **mfhdf**, **df**, **jpeg**), which can, e.g., be obtained through an ANACONDA installation.

2.3.3 Test Suite

After the installation, it is advisable to first execute the test suite of SHARC, which will test the fundamental functionality of SHARC. Change to an empty directory and execute

```
$SHARC/tests.py
```

The interactive script will first verify the Python installation (no message will appear if the Python installation is fine). Subsequently, the script prompts the user to enter which tests should be executed. The script will also ask for a number of environment variables, which are listed in Table ??.

Their is at least one test for each of the auxiliary scripts and interfaces. Tests whose names start with **scripts_** test the functionality of the auxiliary programs in the SHARC suite. Tests whose names start with **ADF_**, **Analytical_**, **COLUMBUS_**, **GAUSSIAN_**, **LVC_**, **MOLCAS_** **MOLPRO_**, or **TURBOMOLE_** run short trajectories, testing whether the main dynamics code, the interfaces, the quantum chemistry programs, and auxiliary programs (THEODORE, WFOVERLAP, ORCA, TINKER) work together correctly.

If the installation was successful and Python is installed correctly, **Analytical_overlap**, **LVC_overlap**, and most tests named **scripts_<NAME>** should execute without error.

The test calculations involving the quantum chemistry programs can be used to check that SHARC can correctly call these programs and that they are installed correctly.

If any of the tests show differences between output and reference output, it is advisable to check the respective files (i.e., compare **$SHARC/../tests/RESULTS/<job>/** to **./RUNNING_TESTS/<job>/**). Note that small differences (different sign of values or small numerical deviations) in the output can already occur when using a different version of the quantum chemistry programs, different compilers, different libraries, or different parallization schemes. It should be noted that along trajectories, these small changes can add up to notably influence the trajectories, but across the ensemble these small changes will likely cancel out.

Table 2.1: Environment variables for SHARC test jobs. These variables need to be set before the test job execution.

Keyword	Description
`$ADF`	Points to the main directory of the ADF installation, which contains the file **`adfrc.sh`** and subdirectory **`bin/`**.
`$BAGEL`	Points to the main directory of the BAGEL installation, which contains subdirectories **`bin/`** and **`lib/`**.
`$COLUMBUS`	Points to the directory containing the COLUMBUS executables, e.g., **`runls`**.
`$GAUSSIAN`	Points to the main directory of the GAUSSIAN installation, which contains the GAUSSIAN executables (e.g., **`g09`**/**`g16`** or **`l9999.exe`**).
`$MOLCAS`	Points to the main directory of the OPENMOLCAS installation, containing **`molcas.rte`** and directories **`basis_library/`** and **`bin/`**.
`$MOLPRO`	Points to the **`bin/`** directory of the MOLPRO installation, which contains the **`molpro.exe`** file.
`$TURBOMOLE`	Points to the main directory of the TURBOMOLE installation, which contains subdirectories like **`basen/`**, **`bin/`**, or **`scripts/`**.
`$ORCA`	Points to the directory containing the ORCA executables, e.g., **`orca`**, **`orca_gtoint`**, or **`orca_fragovl`**.
`$THEODORE`	Points to the main directory of the THEODORE installation. **`$THEODORE/bin/`** should contain **`analyze_tden.py`**.
`$TINKER`	Points to the main directory of a MOLCAS-modified TINKER installation. **`$TINKER/bin`** should contain **`tkr2qm_s`**).
`$PYQUANTE`	Points to the main directory of PYQUANTE, containing the **`PyQuante/`** subdirectory that can be imported by Python as a module.
`$molcas`	Should point to the same location as **`$MOLCAS`**, or another MOLCAS installation. Note that **`$molcas`** is only used by some COLUMBUS test jobs. Also note that **`$molcas`** does not need to point to the MOLCAS installation interfaced to COLUMBUS.
`$orca`	Should point to the same location as **`$ORCA`**, or another ORCA installation. Note that **`$orca`** is only used by some tests where ORCA is used as helper program (e.g., for **`setup_orca_opt.py`** or spin-orbit calculations with **`SHARC_RICC2.py`**).

2.3.4 Additional Programs

For full functionality of the SHARC suite, several additional programs are recommended (all of these programs are currently freely available, except for AMBER):

- The Python package NUMPYhttp://www.numpy.org/NUMPY.
  Optimally, within your Python installation the NUMPY package (which provides many numerical methods, e.g., matrix diagonalization) should be available. If NUMPY is not available, some parts of the SHARC suite are still functional, and the affected scripts will fall back to use a small Fortran code (front-end for LAPACK) within the SHARC package. Since in the Python scripts no large-scale matrix calculations are carried out, there should be no significant performance loss if NUMPY is not available. NUMPY is mandatory for dynamics with the ADF interface, with LVC models, and all **`pysharc`** functionalities.
- The Python package MATPLOTLIBhttps://matplotlib.org/MATPLOTLIB.
  If the MATPLOTLIB package, some auxiliary scripts (**`trajana_nma.py`** and **`trajana_essdyn.py`**) can automatically generate certain plots.
- The GNUPLOThttp://www.gnuplot.info/GNUPLOT plotting software.
  GNUPLOT is not strictly necessary, since all output files could be plotted using other plotting programs. However, a number of scripts from the SHARC suite automatically generate GNUPLOT scripts after data processing, allowing to quickly plot the results.
- A molecular visualization software able to read xyz files (e.g. MOLDENhttp://www.cmbi.ru.nl/molden/molden.htmlMOLDEN, GABEDIThttp://gabedit.sourceforge.net/GABEDIT, MOLEKELhttp://molekel.cscs.ch/wiki/pmwiki.phpMOLEKEL or VMDhttp://www.ks.uiuc.edu/Research/vmd/VMD).
  Molecular visualization software is needed in order to animate molecular motion in the dynamics.
- The THEODOREhttp://theodore-qc.sourceforge.net/THEODORE wave function analysis suite (version 2.0 or higher).

The wave function analysis package THEODORE allows to compute various descriptors of electronic wave functions (supported by some interfaces), which is helpful to follow the state characters along trajectories.

- The AMBERhttp://ambermd.org/AMBER molecular dynamics package.
AMBER can be used to prepare initial conditions based on ground state molecular dynamics simulations (instead of using a Wigner distribution), which is especially useful for large systems.
- The ORCAhttps://orcaforum.kofo.mpg.deORCA ab initio package.
ORCA can be employed as external optimizer. In combination with the SHARC interfaces, it is possible to perform optimizations of minima, conical intersections, and crossing points for any method interfaced to SHARC.

2.3.5 Quantum Chemistry Programs

Even though SHARC comes with two interfaces for analytical potentials (and hence can be used without any quantum chemistry program), the main application of SHARC is certainly on-the-fly ab initio dynamics. Hence, one of the following interfaced quantum chemistry programs is necessary:

- MOLPROhttp://www.molpro.net/MOLPRO (this release was checked against MOLPRO 2010 and 2012).
- OPENMOLCAShttps://gitlab.com/Molcas/OpenMolcas/OPENMOLCAS (this release was checked against OPENMOLCAS 18).
  - TINKERhttp://dasher.wustl.edu/tinker/TINKER, interfaced to OPENMOLCAS 18http://www.molcas.org/documentation/mar
    to OPENMOLCAS 18, for QM/MM dynamics.
- MOLCAShttp://www.molcas.org/MOLCAS (this release was checked against MOLCAS 8.3 and 8.4).
  - TINKERhttp://dasher.wustl.edu/tinker/TINKER, interfaced to MOLCAShttp://www.molcas.org/documentation/manual/nod
    to MOLCAS, for QM/MM dynamics.
- COLUMBUS 7http://www.univie.ac.at/columbus/docs$_COL70/documentation_main.html$COLUMBUS7
  - COLUMBUS-MOLCAS interfacehttp://www.univie.ac.at/columbus/docs$_COL70/columbus_molcas_link.html$COLUMBUS-MOLCA
    for spin-orbit couplings.
- AMSTERDAM DENSITY FUNCTIONALhttp://www.scm.com/ADFAMSTERDAM DENSITY FUNCTIONAL (only ADF 2019 or newer)
- TURBOMOLEhttp://www.turbomole.comTURBOMOLE (this release was checked against TURBOMOLE 6.6, 7.0, 7.1, 7.2, 7.3, and 7.4).
  - ORCAhttps://orcaforum.kofo.mpg.deORCA (version 3 or 4) for providing spin-orbit integrals.
  - TINKERhttp://dasher.wustl.edu/tinker/TINKER (this release was checked against version 6.3.3) for QM/MM dynamics.
- GAUSSIANhttp://www.gaussian.comGAUSSIAN (this release was checked against GAUSSIAN 09 and 16).
- ORCAhttps://orcaforum.kofo.mpg.deORCA (version 4.1 or 4.2).
  - TINKERhttp://dasher.wustl.edu/tinker/TINKER (this release was checked against version 6.3.3) for QM/MM dynamics.
- BAGELhttps://nubakery.org/index.htmlBAGEL (commit 0ea6b59 from Mar 27, 2019 or newer).
  - PyQuantehttp://pyquante.sourceforge.net/PyQuante for overlap calculations

See the relevant sections in chapter **??** for a description of the quantum chemical methods available with each of these programs.

3 Execution

The SHARC suite consists of the main dynamics code **sharc.x** and a number of auxiliary programs, like setup scripts and analysis tools. Additionally, the suite comes with interfaces to several quantum chemistry software: MOLPRO, MOLCAS, COLUMBUS, TURBOMOLE, ADF, GAUSSIAN, ORCA, and BAGEL.

In the following, first it is explained how to run a single trajectory by setting up all necessary input for the dynamics code **sharc.x** manually. Afterwards, the usage of the auxiliary scripts is explained. Detailed infos on the SHARC input files is given in chapter ??. Chapter ?? documents the different output files SHARC produces. The interfaces are described in chapter ?? and the auxiliary scripts in chapter ??. All relevant theoretical background is given in chapter ??.

3.1 Running a single trajectory

3.1.1 Input files

SHARC requires a number of input files, which contain the settings for the dynamics simulation (**input**), the initial geometry (**geom**), the initial velocity (**veloc**), the initial coefficients (**coeff**) and the laser field (**laser**). Only the first two (**input**, **geom**) are mandatory, the others are optional. The necessary files are shown in figure ??. The content of the main input file is explained in detail in section ??, the geometry file is specified in section ??. The specifications of the velocity, coefficient and laser files are given in sections ??, ?? and ??, respectively.

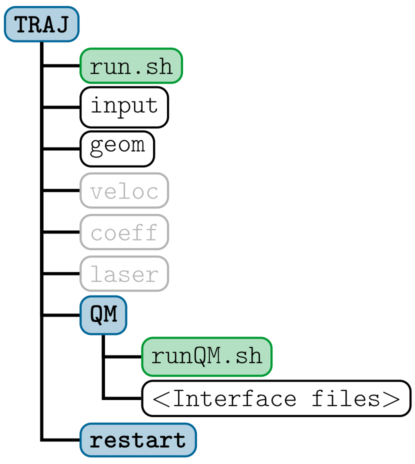

Figure 3.1: Input files for a SHARC dynamics simulation. Directories are in blue, executable scripts in green, regular files in white and optional files in grey.

Additionally, the directory **QM/** and the script **QM/runQM.sh** need to be present, since the on-the-fly ab initio calculations are implemented through these files. The script **QM/runQM.sh** is called each time SHARC performs an on-the-fly calculation of electronic properties (usually by a quantum chemistry program). In order to do so, SHARC first writes the request for the calculation to **QM/QM.in**, then calls **QM/runQM.sh**, waits for the script to finish and then reads the requested quantities from **QM/QM.out**. The script **QM/runQM.sh** is fully responsible to generate the requested results from the provided input. In virtually all cases, this task is handled by the SHARC-quantum chemistry interfaces (see chapter ??), so that the script **QM/runQM.sh** has a particularly simple form:

```
cd QM/
$SHARC/<INTERFACE> QM.in
```

with the corresponding interface name given. Note that the interfaces in all cases need additional input files, which must be present in **QM/**. Those input files contain the specifications for the quantum chemistry information, e.g., basis set, active and reference space, memory settings, path to the quantum chemistry program, path to scratch directories; or for **SHARC_Analytical.py** and **SHARC_LVC.py** the parameters for the analytical potentials. For each interface, the input files are slightly different. See sections **??**, **??**, **??**, **??**, **??**, **??**, **??**, **??**, **??**, or **??** for the necessary information.

3.1.2 Running the dynamics code

Given the necessary input files, SHARC can be started by executing

```
user@host> $SHARC/sharc.x input
```

Note that besides the input file, at least the geometry file needs to be present (see chapter **??** for details).

A running trajectory can be stopped after the current time step by creating an empty file **STOP**:

```
user@host> touch STOP
```

This is usually preferable to simply killing SHARC, because the current time step is properly finished and all files are correctly written for analysis and restart.

3.1.3 Output files

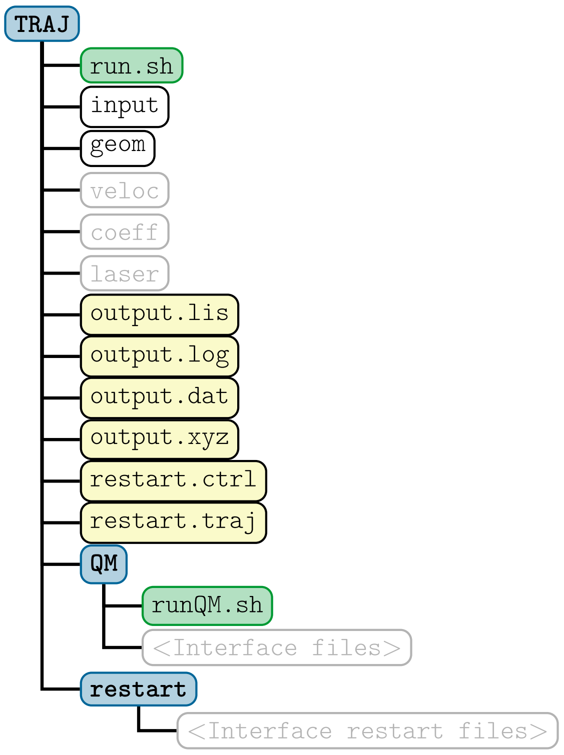

Figure 3.2: Files of a SHARC dynamics simulation after running. Directories are in blue, executable scripts in green, regular files in white and optional files in grey. Output files are in yellow.

Figure ?? shows the content of a trajectory directory after the execution of SHARC. There will be six new files. These files are **output.log**, **output.lis**, **output.dat** and **output.xyz**, as well as **restart.ctrl** and **restart.traj**.

The file **output.log** contains mainly a listing of the chosen options and the resulting dynamics settings. At higher print levels, the log file contains also information per time step (useful for debugging). **output.lis** contains a table with one line per time step, giving active states, energies and expectation values. **output.dat** contains a list of all important matrices and vectors at each time step. This information can be extracted with **data_extractor.x** to yield plottable table files. **output.xyz** contains the geometries of all time steps (the comments to each geometry give the active state). For details about the content of the output files, see chapter ??.

The restart files contain the full state of a trajectory and its control variables from the last successful time step. These files are needed in order to restart a trajectory at this time step (either because the calculation failed, or in order to extend the simulation time beyond the original maximum simulation time). The **restart/** directory contains all persistent files that are needed for the quantum chemistry interfaces (for restarting, but also between all time steps).

3.2 Typical workflow for an ensemble of trajectories

Usually, one is not interested in running only a single trajectory, since a single trajectory cannot reproduce the branching of a wave packet into different reaction channels. In order to do so, within surface hopping an ensemble of independent trajectories is employed.

When dealing with a (possibly large) ensemble of trajectories, setup and analysis need to be automatized. Hence, the SHARC suite contains a number of scripts fulfilling different tasks in the usual workflow of setting up ensembles of trajectories. The typical workflow is given schematically in figure ??.

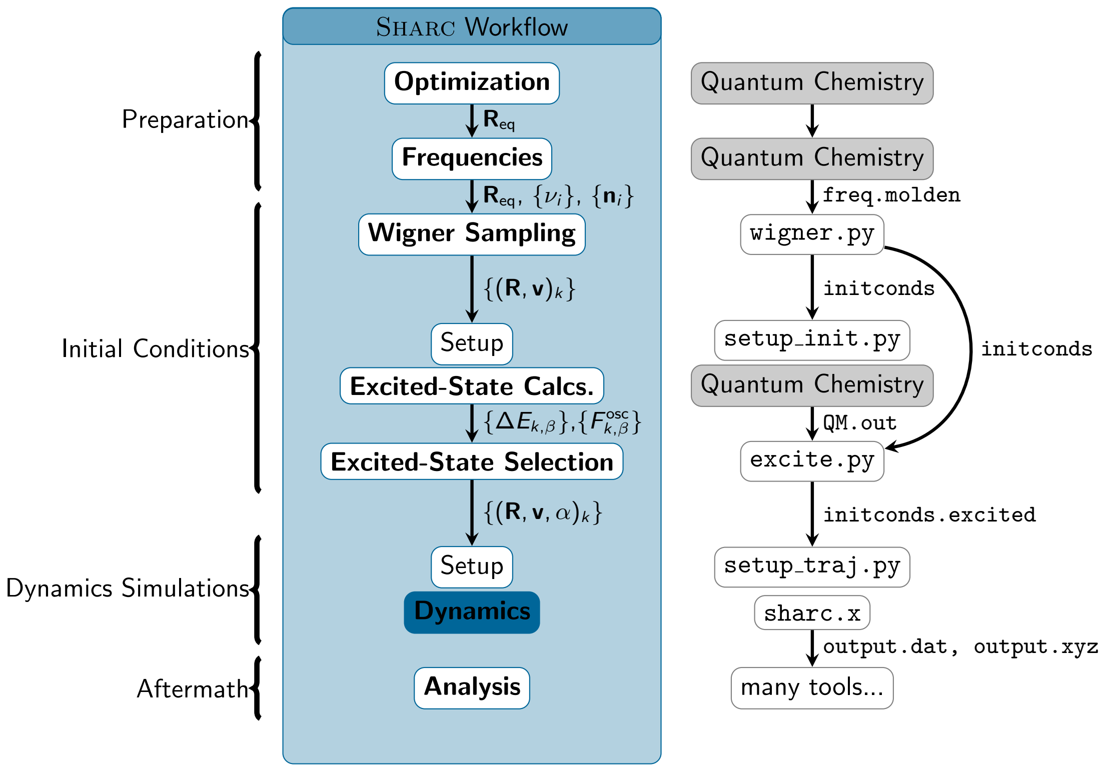

Figure 3.3: Typical workflow for conducting excited-state dynamics simulations with SHARC.

3.2.1 Initial condition generation

In the typical workflow, the user will first create a set of suitable initial conditions. In the context of the SHARC package, an initial condition is a set of an initial geometry, initial velocity, initial occupied state and initial wave function coefficients. Many such sets are needed in order to setup physically sound dynamics simulations.

Generation of initial geometries and velocities Currently, within the SHARC suite, initial geometries and velocities can be generated based on a quantum harmonic oscillator Wigner distribution. The theoretical background is given in section ??. The calculation is performed by **`wigner.py`**, which is explained in section ??.

As given in figure ??, **`wigner.py`** needs as input the result of a frequency calculation in MOLDEN format. The calculation can be performed by any quantum chemistry program and any method the user sees fit (there are some scripts which can aid in the frequency calculation, see sections ??, ??, ??). **`wigner.py`** produces the file **`initconds`**, which contains a list of initial conditions ready for further processing.

Alternatively, initial geometries and velocities can be extracted from molecular dynamics simulations in the ground state. Currently, it is possible to either convert restart files AMBER to an **`initconds`** file (using **`amber_to_initconds.py`**, see section ??) or to randomly sample snapshots from SHARC trajectories (using **`sharctraj_to_initconds.py`**, see section ??).

Generation of initial coefficients and states In the second preparation step, for each of the sampled initial geometries it has to be decided which excited state should be the initial one. In simple cases, the user may manually choose the initial excited state using **`excite.py`** (optionally after diabatization; see ??). Alternatively, the selection of initial states can be performed based on the excitation energies and oscillator strengths of the excited states at each initial geometry (this approximately simulates a delta-pulse excitation).

The latter options (diabatization or energies/oscillator strengths) make it necessary to carry out vertical excitation calculation before the selection of the initial states. The calculations can be set up with **`setup_init.py`** (see section ??). This script prepares for each initial condition in the **`initconds`** file a directory with the necessary input to perform the calculation. The user should then execute the run script (**`run.sh`**) in each of the directories (either manually or through a batch queueing system).

After the vertical excitation calculations are completed, the vertical excitation energies and oscillator strengths of each calculation are collected by **`excite.py`** (see ??). The same script then performs the selection of the initial electronic state for each initial geometry. The results are written to a new file, **`initconds.excited`**. This file contains all information needed to setup the ensemble.

Additionally, **`spectrum.py`** (??) can calculate absorption spectra based on the **`initconds.excited`** file. This may be useful to verify that the level of theory chosen is appropriate (e.g., by comparing to an experimental spectrum), or to choose a suitable excitation window for the determination of the initial state.

3.2.2 Running the dynamics simulations

Based on the initial conditions given in **`initconds.excited`**, the input for all trajectories in the ensemble can be setup by **`setup_traj.py`** (see section ??). The script produces one directory for each trajectory, containing the input files for SHARC and the selected interface.

In order to run a particular trajectory, the user should execute the run script (**`run.sh`**) in the directory of the trajectory. Since those calculations can run between minutes and several weeks (depending on the level of theory used and the number of time steps), it is advisable to submit the run scripts to a batch queueing system.

The progress of the simulations can be monitored most conveniently in the **`output.lis`** files. If the calculations are running in some temporary directory, the output files can be copied to the local directory (where they were setup) with the **`scp`** wrapper **`retrieve.sh`** (see ??). This allows to perform ensemble analysis while the trajectories are still running.

If a trajectory fails, the temporary directory where the calculation is running is not deleted. The file **`README`** will be created in the trajectory's directory, giving the time of the failure and the location of the temporary data, so that the case can be investigated.

In order to signal SHARC to terminate a trajectory after the current time step is completed, the user can create a (possibly empty) file **`STOP`** in the working directory of the trajectory (the directory where **`sharc.x`** is running).

The status of the ensemble of trajectories can be checked with **`diagnostics.py`** (section ??). This script checks the presence and integrity of all relevant files, the progress of all trajectories, and warns if trajectories behave unexpectedly (non-conversion of total energy, intruder states, etc).

3.2.3 Analysis of the dynamics results

Each trajectory can be analyzed independently by inspecting the output files (see chapter ??). Most importantly, calling **`data_extractor.x`** (??) on the **`output.dat`** file of a trajectory creates a number of formatted files which can be plotted with the help of **`make_gnuscript.py`** (??) and GNUPLOT. The nuclear geometries in **`output.xyz`** file can be analyzed in terms of internal coordinates (bond lengths, angles, ring conformations, etc.) using **`geo.py`** (??). The manual analysis of all individual trajectories is usually a good idea to verify that the trajectories are correctly executing, and in order to find general reaction pathways. The manual analysis often permits to formulate some hypotheses, which can then be verified with the statistical analysis tools.

For the complete ensemble, the first step should usually be to run **`diagnostics.py`** (section ??). This script will determine how long the different trajectories are, and, more importantly, will check the trajectories for file integrity, conservation of total energy, and continuity of potential/kinetic energy. Based on a set of customizable criteria, the script determines for each trajectory a "maximum usable time". The script then can mark all trajectories with maximum usable time below a given threshold to be excluded from analysis (by creating a file **`DONT_ANALYZE`** in the trajectory's directory). The other analysis scripts will then ignore trajectories marked by **`diagnostics.py`**. Trajectories can also be manually excluded from analysis, by creating a file called **`CRASHED`**, **`DEAD`**, or **`RUNNING`** in the respective directory.

After the trajectories were checked and unsuitable ones excluded, the statistical analysis scripts can be used. The script **`populations.py`** (section ??) can calculate average excited-state populations. The script **`transition.py`** (section ??) can analyze the total number of hops between all pairs of states in an ensemble, allowing to identify relevant relaxation routes in the dynamics. Using the script **`make_fit.py`** (??), it is possible to make elaborate global fits of chemical kinetics models to the populations data, allowing to extract rate constants from the populations, and to compute errors for these rate constants. The same functionality is also present in the older scripts **`make_fitscript.py`** (??) and **`bootstrap.py`** (??). The script **`crossing.py`** (??) can find and extract notable geometries, e.g., those geometries where a surface hop between two particular states occurred. Using **`trajana_essdyn.py`** (??) and **`trajana_nma.py`** (??) it is furthermore possible to perform essential dynamics analysis and normal mode analysis. Finally, **`data_collector.py`** can merge arbitrary tabulated data from the trajectories and perform various analysis procedures (compute mean/standard deviation, data convolution, summation, integration), which can be used to compute, e.g., time-dependent distribution functions or time-dependent spectra.

3.3 Auxilliary Programs and Scripts

The following tables list the auxiliary programs in the SHARC suite. The rightmost column gives the section where the program is documented.

3.3.1 Setup

`wigner.py`	Creates initial conditions from Wigner distribution.	??
`amber_to_initconds.py`	Creates initial conditions from Amber restart files.	??
`sharctraj_to_initconds.py`	Creates initial conditions from SHARC trajectories.	??
`setup_init.py`	Sets up initial vertical excitation calculations.	??
`excite.py`	Generates excited state lists for initial conditions and selects initial states.	??
`setup_traj.py`	Sets up the dynamics simulations based on the initial conditions.	??
`laser.x`	Prepares files containing laser fields.	??
`molpro_input.py`	Prepares MOLPRO input files and template files for the MOLPRO interface.	??
`molcas_input.py`	Prepares MOLCAS input files and template files for the MOLCAS interface.	??
`ADF_input.py`	Prepares ADF input files.	??
`ADF_freq.py`	Converts ADF output files of frequency calculations to Molden format.	??
`setup_LVCparam.py`	Sets up single point calculations for LVC parametrization.	??
`create_LVCparam.py`	Produces LVC model files from parametrization data.	??

3.3.2 Analysis

`spectrum.py`	Generates absorption spectra from initial conditions files.	??
`retrieve.sh`	**`scp`** wrapper to retrieve dynamics output during the simulation.	??
`data_extractor.x`	Extracts plottable results from the SHARC output data file.	??
`data_extractor_NetCDF.x`	Extracts plottable results from the SHARC output data file in NetCDF format.	??
`data_converter.x`	Converts **`output.dat`** files to **`output.dat.nc`** files.	??
`make_gnuscript.py`	Creates gnuplot scripts to plot trajectory data.	??
`diagnostics.py`	Checks ensembles for integrity, progress, energy conservation.	??
`populations.py`	Calculates ensemble populations.	??
`transition.py`	Calculates total number of hops within an ensemble.	??
`make_fit.py`	Performs kinetic model fits and bootstrapping.	??
`make_fitscript.py`	Creates GNUPLOT scripts to fit kinetic models to population data.	??
`bootstrap.py`	Computes errors for kinetic model fits.	??
`crossing.py`	Extracts specific geometries from ensembles.	??
`geo.py`	Calculates internal coordinates from xyz files.	??
`trajana_essdyn.py`	Performs an essential dynamics analysis for an ensemble.	??
`trajana_nma.py`	Performs a normal mode analysis for an ensemble.	??
`data_collector.py`	Collects data from tabular files and performs various analyses	??

3.3.3 Interfaces

`SHARC_MOLPRO.py`	Calculates SOCs, gradients, nonadiabatic couplings, overlaps, dipole moments, and Dyson norms at the CASSCF level of theory (using MOLPRO). Symmetry or RASSCF are not supported. Only segmented basis sets are possible.	??
`SHARC_MOLCAS.py`	Calculates SOCs, gradients, overlaps, dipole moments, Dyson norms, dipole moment derivatives, and spin-orbit coupling derivatives at the CASSCF and (MS-)CASPT2 level of theory (using MOLCAS). Symmetry is not supported. Numerical differentiation is used for some tasks.	??
`SHARC_COLUMBUS.py`	Calculates SOCs, gradients, nonadiabatic couplings, overlaps, dipole moments, and Dyson norms at the CASSCF, RASSCF, and MRCISD levels of theory (using COLUMBUS). Symmetry is not supported. Works with the either DALTON or SEWARD integrals (through the COLUMBUS-MOLCAS interface), but SOCs are only available with SEWARD integrals. Nonadiabatic couplings are only available with DALTON integrals.	??
`SHARC_Analytical.py`	Calculates SOCs, gradients, overlaps, dipole moments, and dipole moment derivatives based on analytical expressions of diabatic matrix elements defined in Cartesian coordinates.	??
`SHARC_ADF.py`	Calculates SOCs, gradients, overlaps, dipole moments, and Dyson norms at the TD-DFT level of theory with GGA and hybrid functionals (using ADF). Symmetry is not supported.	??
`SHARC_RICC2.py`	Calculates SOCs, gradients, overlaps, and dipole moments at the ADC(2) and CC2 levels of theory (using TURBOMOLE). Symmetry is not supported. For SOCs, only ADC(2) can be used and ORCA has to be installed in addition to TURBOMOLE.	??
`SHARC_LVC.py`	Calculates SOCs, gradients, nonadiabatic couplings, overlaps, and dipole moments based on linear-vibronic coupling models defined in mass-weighted normal mode coordinates.	??
`SHARC_GAUSSIAN.py`	Calculates gradients, overlaps, dipole moments, and Dyson norms at the TD-DFT level of theory (using Gaussian). Symmetry is not supported.	??
`SHARC_ORCA.py`	Calculates SOCs, gradients, overlaps, dipole moments, and Dyson norms at the TD-DFT level of theory with GGA and hybrid functionals (using ORCA). Symmetry is not supported.	??
`SHARC_BAGEL.py`	Calculates gradients, nonadiabatic couplints, overlaps, dipole moments, and Dyson norms at the CASSCF, CASPT2, MS-CASPT2, and XMS-CASPT2 level of theory (using BAGEL). Symmetry is not supported.	??

3.3.4 Others

`tests.py`	Script to automatically run the SHARC test suite.	??
`wfoverlap.x`	Program to compute wave function overlaps, used by most interfaces.	??
`Orca_External`	Script to carry out optimizations with ORCA as optimizer and SHARC as gradient provider.	??
`setup_orca_opt.py`	Script to setup optimizations with ORCA as optimizer and SHARC as gradient provider.	??
`setup_single_point.py`	Script to setup single point calculations with SHARC interfaces.	??
`QMout_print.py`	Script to convert a **`QM.out`** file to a table with energies and oscillator strengths.	??
`diagonalizer.x`	Helper program for **`excite.py`**. Only required if NUMPY is not available.	??

3.4 The PYSHARC dynamics driver

Since SHARC2.1, the suite contains alternative dynamics drivers, besides the regular **`sharc.x`** executable, which are based on the **`pysharc`** sub-project. **`pysharc`** is an implementation of the SHARC dynamics driver in Python that directly calls the Fortran routines of **`sharc.x`** (through the Python-C API via intermediate C routines). In this way, both the **`sharc.x`** code and the quantum chemistry interface code can be run within the same program, omitting file-based communication between driver and quantum chemistry interface.

The motivation of this is that the omission of file-based communication can save up to a few seconds in each time step. Naturally, this is negligible for expensive quantum chemistry where a time step takes several minutes. However, for very fast interfaces (analytical models, LVC models), time steps can be reduced from a few seconds to a few milliseconds, providing a dramatic speedup of the SHARC simulations which is only limited by the speed of writing the output files. Consequently, the use of **`pysharc`** is highly advisable for SHARC simulations with very fast interfaces. Currently, **`pysharc`** is only implemented for LVC models, through the driver script named **`pysharc_lvc.py`**.

`pysharc` is intended to change the SHARC workflow as little as possible. Basically all features of **`sharc.x`** are supported. The only change that needs to be applied is to call the **`pysharc`** driver instead of **`sharc.x`**.

The high efficiency of **`pysharc`** is optimally combined with an efficient way of producing output. As the default ASCII output format is not optimized for performance, together with **`pysharc`** SHARC2.1 contains an implementation that produces NetCDF binary output data files. It is advisable that every time **`pysharc`** is run, output is written in NetCDF format. Details of NetCDF output are documented in Section **??**

4 Input files

In this chapter, the format of all SHARC input files are presented. Those are the main input file (here called **input**), the geometry file, the velocity file, the coefficients file, the laser file, and the atom mask file. Only the first two are mandatory, the others are optional input files. All input files are ASCII text files.

4.1 Main input file

This section presents the format and all input keywords for the main SHARC input. Note that when using **setup_traj.py**, full knowledge of the SHARC input keywords is not required.

4.1.1 General remarks

The input file has a relatively flexible structure. With very few exceptions, each single line is independent. An input line starts with a keyword, followed optionally by a number of arguments to this keyword. Example:

```
stepsize 0.5
```

Here, **stepsize** is the keyword, referring to the size of the time steps for the nuclear motion in the dynamics. **0.5** gives the size of this time step, in this example 0.5 fs.

A number of keywords have no arguments and act as simple switches (e.g., **restart**, **gradcorrect**, **grad_select**, **nac_select**, **ionization**, **track_phase**, **dipole_gradient**). Those keywords can be prefixed with **no** to explicitly deactivate the option (e.g., **norestart** deactivates restarts).

In each line a trailing comment can be added in the input file, by using the special character **#**. Everything after **#** is ignored by the input parser of SHARC. The input file also can contain arbitrary blank lines and lines containing only comments. All input is case-insensitive.

The input file is read by SHARC by subsequently searching the file for all known keywords. Hence, unknown or misspelled keywords are ignored. Also, the order of the keywords is completely arbitray. Note however, that if a keyword is repeated in the input only the *first* instance is used by the program.

4.1.2 Input keywords

In Table ??, all input keywords for the SHARC input file are listed.

Table 4.1: Input keywords for **`sharc.x`**. The first column gives the name of the keyword, the second lists possible arguments and the third line provides an explanation. Defaults are marked like **this**. $\$n$ denotes the n-th argument to the keyword.

Keyword	Arguments	Explanation
— General control keywords —		
`printlevel`	**integer**	Controls the verbosity of the log file.
	$1=0	Log file is empty
	$1=1	+ List of internal steps
	$1=**2**	+ Input parsing information
	$1=3	+ Some information per time step
	$1=4	+ More information per time step
	$1=5	+ Much more information per time step
`restart`		Dynamics is resumed from restart files.
`norestart`		Dynamics is initialized from input files.
		`norestart` takes precedence.
`rngseed`	**integer**	Seed for the random number generator.
	10997279	Used for surface hopping and AFSSH decoherence.
`compatibility`	$1=**0**	Compatibility mode disabled.
	$1=1	Do not draw a second random number per step (for decoherence).
— Input file keywords —		
`geomfile`	**quoted string**	File name containing the initial geometry.
	"geom"	
`velocfile`	**quoted string**	File containing the initial velocities.
	"veloc"	Only read if **`veloc external`**.
`coefffile`	**quoted string**	File containing the initial wave function coefficients.
	"coeff"	Only read if **`coeff external`**.
`laserfile`	**quoted string**	File containing the laser field.
	"laser"	Only read if **`laser external`**.
`atommaskfile`	**quoted string**	File containing the atom mask.
	"atommask"	Only read if **`atommask external`**.
— Trajectory initialization keywords —		
`veloc`	**string**	Sets the initial velocities.
	$1=**zero**	Initial velocities are zero.
	$1=random $2 **float**	Random initial velocities with $2 eV kinetic energy per atom.
	$1=external	Initial velocities are read from file.
`nstates`	list of **integer**s	Number of states per multiplicity.
	$1 (**1**)	Number of singlet states
	$2 (**0**)	Number of doublet states
	$3 (**0**)	Number of triplet states
	$... (**0**)	Number of states of higher multiplicities
`actstates`	list of **integer**s	Number of active states per multiplicity.
	`same as nstates`	By default, all states are active.
`state`	**integer**, **string**	Specifies the initial state. (no default; SHARC exits if **`state`** is missing).
	$1	Initial state.
	$2=MCH	Initial state and coefficients are given in MCH representation.
	$2=diag	Initial state and coefficients are given in diagonal representation.
`coeff`	**string**	Sets the wave function coefficients.
	$1=**auto**	Initial coefficient are determined automatically from initial state.
	$1=external	Initial coefficients are read from file.
— Laser field keywords —		
`laser`	**string**	Sets the laser field.
	$1=**none**	No laser field is applied.

Continued on next page

Table 4.1 – Continued from previous page

Keyword	Arguments	Explanation
	\$1=internal \$1=external	Laser field is calculated at each time step from internal function. Laser field for each time step is read during initialization.
`laserwidth`	**float** **1.0 eV**	Laser bandwidth used to detect induced hops.
		— Time step keywords —
`stepsize`	**float** **0.5 fs**	Length of the nuclear dynamics time steps in fs.
`nsubsteps`	**integer** **25**	Number of substeps for the integration of the electronic equation of motion.
`nsteps`	**integer** **3**	Number of simulation steps.
`tmax`	**float**	Total length of the simulation in fs. No effect if **`nsteps`** is present.
`killafter`	**float** **-1**	Terminates the trajectory after \$1 fs in the lowest state. If \$1<0, trajectories are never killed.
		— Surface hopping setting keywords —
`surf`	**string** \$1=**diagonal,sharc** \$1=MCH	Potential energy surfaces used in surface hopping. Uses diagonal potentials. Uses MCH potentials.
`coupling`	**string** \$1=ddr,nacdr \$1=ddt,nacdt \$1=**overlap**	Quantities describing the nonadiabatic couplings. Uses vectorial nonadiabatic couplings $\langle\psi_\alpha\|\partial/\partial R\|\psi_\beta\rangle$. Uses temporal nonadiabatic couplings $\langle\psi_\alpha\|\partial/\partial t\|\psi_\beta\rangle$. Uses the overlaps $\langle\psi_\alpha(t_0)\|\psi_\beta(t)\rangle$ (local diabatization).
`gradcorrect` **`nogradcorrect`**		Include $(E_\alpha - E_\beta)\langle\psi_\alpha\|\partial/\partial R\|\psi_\beta\rangle$ in gradient transformation. Transform only the gradient matrix.
`ekincorrect`	**string** \$1=none \$1=**parallel_vel** \$1=parallel_nac \$1=parallel_diff	Adjustment of the kinetic energy after a surface hop. Kinetic energy is not adjusted. Jumps are never frustrated. Velocity is rescaled to adjust kinetic energy. Only the velocity component in the direction of $\langle\psi_\alpha\|\partial/\partial R\|\psi_\beta\rangle$ is rescaled. Only the velocity component in the direction of $\Delta\nabla E$ is rescaled.
`reflect_frustrated`	**string** \$1=**none** \$1=parallel_vel \$1=parallel_nac \$1=parallel_diff	Reflection of trajectory after frustrated hop. No reflection. Full velocity vector is reflected. Only the velocity component in the direction of $\langle\psi_\alpha\|\partial/\partial R\|\psi_\beta\rangle$ is reflected. Only the velocity component in the direction of $\Delta\nabla E$ is reflected.
`decoherence_scheme`	**string** \$1=**none** \$1=edc \$1=afssh	Method for decoherence correction. No decoherence correction. Energy-difference based correction.[?] Augmented FSSH.[?]
`decoherence_param`	**float** **0.1**	Value α in EDC decoherence (in Hartree). \$1> 0.0
`decoherence` **`nodecoherence`**		Applies decoherence correction (EDC by default). No decoherence correction. **`nodecoherence`** takes precedence.
`hopping_procedure`	**string** \$1=off \$1=**sharc,standard** \$1=gfsh	Method for hopping probabilities. No hops (same as **`no_hops`**). Standard SHARC hopping probabilities. Global flux SH hopping probabilities.[?]
`no_hops`		Disables surface hopping. **`no_hops`** takes precedence over **`hopping_procedure`**.
`force_hop_to_gs`	**float**	Activates forced hops to lowest state. hop is forced if lowest–active energy difference < \$1

Continued on next page

Table 4.1 – Continued from previous page

Keyword	Arguments	Explanation
`atommask`	**string** $1=**none** $1=external	Activates masking of atoms (for EDC, **`parallel_vel`**). No atoms are masked. Atom mask is read from external file.
— Energy control keywords —		
`ezero`	**float** **0.0**	Energy shift for Hamiltonian diagonal elements (Hartree). Is not determined automatically!
`scaling`	**float** **1.0**	Scaling factor for Hamiltonian matrix and gradients. 0. <$1
`dampeddyn`	**float** **1.0**	Scaling factor for kinetic energy at each time step. 0. ≤$1≤ 1.
— Gradient and NAC selection keywords —		
`grad_select` **`grad_all`**		Only some gradients are calculated at every time step. All gradients are calculated at every time step (Alias: **`nograd_select`**). **`grad_all`** takes precedence.
`nac_select` **`nac_all`**		Only some $\langle\psi_\alpha\|\partial/\partial R\|\psi_\beta\rangle$ are calculated at every time step. All $\langle\psi_\alpha\|\partial/\partial R\|\psi_\beta\rangle$ are calculated at every time step (Alias: **`nonac_select`**). **`nac_all`** takes precedence.
`eselect`	**float** **0.5 eV**	Parameter for selection of gradients and NACs (in eV).
`select_directly` `noselect_directly`		Do not do a second QM calculation for gradients and NACs. Do a second QM calculation for gradients and NACs.
— Phase tracking keywords —		
`track_phase` `notrack_phase`		Track the phase of the transformation matrix **U**. No phase tracking of **U** (can provide very large speedups in **`pysharc`**, but check whether results are affected).
`phases_from_interface` `nophases_from_interface`		Request phase information from interface. Try to recover phase information from QM data.
`phases_at_zero`		Request phase from interface at $t = 0$.
— Property computation keywords —		
`spinorbit` `nospinorbit`		Include spin-orbit couplings into the Hamiltonian. Neglect spin-orbit couplings.
`dipole_gradient` **`nodipole_gradient`**		Include dipole moments derivatives in the gradients. Neglect dipole moments derivatives.
`ionization` **`noionization`**		Calculate ionization probabilities on-the-fly. No ionization probabilities.
`ionization_step`	**integer** **1**	Calculate ionization probabilities every $1 time step. By default calculated every time step (if **`ionization`**).
`theodore` **`notheodore`**		Calculate wavefunction descriptors on-the-fly. No wavefunction descriptors.
`theodore_step`	**integer** **1**	Calculate wavefunction descriptors every $1 time step. By default calculated every time step (if **`theodore`**).
`n_property1d`	**integer** **1**	Allocate for that many 1D properties.
`n_property2d`	**integer** **1**	Allocate for that many 2D properties.
— Output control keywords —		
`write_grad` **`nowrite_grad`**		Writes gradients to **`output.dat`**.
`write_nacdr` **`nowrite_nacdr`**		Writes NACs to **`output.dat`**.
`write_overlap`		Writes overlaps to **`output.dat`**.

Continued on next page

Table 4.1 – Continued from previous page

Keyword	Arguments	Explanation
`nowrite_overlap`		Not written if not requested.
`write_property1d` `nowrite_property1d`	**on if theodore**	Writes 1D properties to **`output.dat`**.
`write_property2d` `nowrite_property2d`	**on if ionization**	Writes 2D properties to **`output.dat`**.
`output_dat_steps`	**integer**	Determines at which steps **`sharc.x`** writes to **`output.dat`**.
	$1 (**1**)	Stride for writing to **`output.dat`** (default every step).
	$2 (**not given**)	Step at which stride is switched to $3.
	$3 (**not given**)	New stride applied if step ≥ $2.
	$4 (**not given**)	Step at which stride is switched to $5.
	$5 (**not given**)	New stride applied if step ≥ $4.
		Always give 1, 3, or 5 arguments.
`output_format`	**string**	Format of **`output.dat`**.
	$1=**ascii**	as documented in section ??.
	$1=netcdf	as documented in section ??.
		Also deactivates writing of restart files (except last step) and **`output.xyz`**.

4.1.3 Detailed Description of the Keywords

Printlevel The **`printlevel`** keyword controls the verbosity of the log file. The data output file (**`output.dat`**) and the listing file (**`output.lis`**) are not affected by the print level. The print levels are described in section ??.

Restart There are two keywords controlling trajectory restarting. The keyword **`restart`** enables restarting, while **`norestart`** disables restart. If both keywords are present, **`norestart`** takes precedence. The default is no restart.

When restarting, all control variables are read from the restart file instead of the input file. The only exceptions are **`nsteps`** and **`tmax`**. In this way, a trajectory which ran for the full simulation time can easily be restarted to extend the simulation time.

Note that none of the auxiliary scripts adds the **`restart`** keyword to the input file. The user has to manually add the restart file to the input files of the relevant trajectories.

RNG Seed The RNG seed is used to initialize the random number generator, which provides the random numbers for the surface hopping procedure (and the AFSSH decoherence scheme). For details how the seed is used internally, see section ??.

Note that in the case of a restart, the random number generator is seeded normally, and then the appropriate number of random numbers is drawn so that the random number sequence is consistent.

Compatibility Mode With this keyword, one can activate a compatibility mode that allows reproducing results of older versions of SHARC. A value of 0 disables compatibility mode.

Currently, the only other option is a value of 1. In this mode, **`sharc.x`** draws only one random number for each step, like it was the case in SHARC 1.0 (from SHARC 2.0, it draws two random numbers per step, one for hopping and one for decoherence schemes). This compatibility mode cannot be combined with the A-FSSH decoherence correction.

Geometry Input The initial geometry must be given in a second file in the input format also used by COLUMBUShttp://www.univie.ac.at/columbus/docs$_COL70/documentation_main.htmlinputformatalsousedby$COLUMBUS. The default name for this file is **`geom`**. The geometry filename can be given in the input file with the **`geomfile`** keyword. Note that the filename has to be enclosed in single or double quotes. See section ?? for more details.

Velocity Input Using the **`veloc`** keyword, the initial velocities can be either set to zero, determined randomly or read from a file. Random determination of the velocities is such that each atom has the same kinetic energy, which must be specified after **`veloc random`** in units of eV. Determination of the random velocities is detailed in ??. Note that after the initial velocities are generated, the RNG is reseeded (i.e., the sequence of random numbers in the surface hopping procedure is independent of whether random initial velocities are used).

Alternatively, the initial velocities can be read from a file. The default velocity filename is `veloc`, but the filename can be specified with the `velocfile` keyword. Note that the filename has to be enclosed in single or double quotes. The file must contain the Cartesian components of the velocity for each atom on a new line, in the same order as in the geomety file. The velocity is interpreted in terms of atomic units (bohr/atu). See section **??** for more details.

Number of States and Active States The keyword `nstates` controls how many states are taken into account in the dynamics. The keyword arguments specify the number of singlet, doublet, triplet, etc. states. There is no hard-coded maximum multiplicity in the Sharc code, however, some interfaces may restrict the maximum multiplicity.

Using the `actstates` keyword, the dynamics can be restricted to some lowest states in each multiplicity. For each multiplicity, the number of active states must not be larger than the number of states. All couplings between the active states and the frozen states are deleted. These couplings include off-diagonal elements in the H^{MCH} matrix, in the overlap matrix, and in the matrix containing the nonadiabatic couplings. Freezing states can be useful if transient absorption spectra are to be calculated without increasing computational cost due to the large number of states.

Note that the initial state must not be frozen.

Initial State The initial state can be given either in MCH or diagonal representation. The keyword `state` is followed by an integer specifying the initial state and either the string `mch` or `diag`. For the MCH representations, states are enumerated according to the canonical state ordering, see **??**. The diagonal states are ordered according to energy. Note that the initial state must be active.

If the initial state is given in the MCH basis but the dynamics is carried out in the diagonal basis, determination of the initial diagonal state is carried out after the initial QM calculation.

Initial Coefficients The initial coefficients can be determined automatically from the initial state, using `coeff auto` in the input file. If the initial state is given in the diagonal representation as i, the initial coefficients are $c_j^{\mathrm{diag}} = \delta_{ij}$. If the initial state is, however, given in the MCH representation, then $c_j^{\mathrm{MCH}} = \delta_{ij}$ and the determination of $\mathbf{c}^{\mathrm{diag}} = \mathbf{U}^{\dagger}\mathbf{c}^{\mathrm{MCH}}$ is carried out after the initial QM calculation. Currently, `coeff auto` is always used by the automatic setup scripts.

Besides automatic determination, the initial coefficients can be read from a file. The default filename is `coeff`, but the filename can be given with the keyword `coefffile`. Note that the filename has to be enclosed in single or double quotes. The file must contain the real and imaginary part of the initial coefficients, one line per state with no blank lines inbetween. These coefficients are interpreted to be in the same representation as the initial state, i.e. the `state` keyword influences the initial coefficients. For details on the file format, see section **??**. Note that the setup scripts currently cannot setup trajectories with `coeff external`, so this can be considered an expert option.

Laser Input The keyword `laser` controls whether a laser field is included in the dynamics (influencing the coefficient propagation and the energies/gradients by means of the Stark effect).

The input of an external laser field uses the file `laser`. This file is specified in **??**.

In order to detect laser-induced hops, Sharc compares the instantaneous central laser energy with the energy gap between the old and new states. If the difference between the laser energy and the energy gap is smaller than the laser bandwidth (given with the `laserwidth` keyword), the hop is classified as laser-induced. Those hops are never frustrated and the kinetic energy is not scaled to preserve total energy (instead, the kinetic energy is preserved).

Simulation Timestep The keyword `stepsize` controls the length of a time step (in fs) for the dynamics. The nuclear motion is integrated using the Velocity-Verlet algorithm with this time step. Surface hopping is performed once per time step and 1–3 quantum chemistry calculations are performed per time step (depending on the selection schedule). Each time step is divided in `nsubsteps` substeps for the integration of the electronic equation-of-motion. Since integration is performed in the MCH representation, the default of 25 substeps is usually sufficient, even if very small potential couplings are encountered. A larger number of substeps might be necessary if high-frequency laser fields are included or if the energy shift (`ezero`) is not well-chosen.

Simulation Time The keyword **`nsteps`** controls the total length of the simulation. The total simulation time is **`nsteps`** times **`stepsize`**. **`nsteps`** does not include the initial quantum chemistry calculation. Instead of the number of steps, the total simulation time can be given directly (in fs) using the keyword **`tmax`**. In this case, **`nsteps`** is calculated as **`tmax`** divided by **`stepsize`**. If both keywords (**`nsteps`** and **`tmax`**) are present, **`nsteps`** is used. All setup scripts will generally use the **`tmax`** keyword.

Using the keyword **`killafter`**, the dynamics can be terminated before the full simulation time. **`killafter`** specifies (in fs) the time the trajectory can move in the lowest-energy state before the simulation is terminated. By default, simulations always run to the full simulation time and are not terminated prematurely.

Surface Treatment The keyword **`surf`** controls whether the dynamics runs on diagonal potential energy surfaces (which makes it a SHARC simulation) or on the MCH PESs (which corresponds to a spin-diabatic [?] or FISH [?] simulation, or a regular surface hopping simulation). Internally, dynamics on the MCH potentials is conducted by setting the U matrix equal to the unity matrix at each time step.

Description of Non-adiabatic Coupling The code allows propagating the electronic wave function using three different quantities describing nonadiabatic effects, see ??. The keyword **`coupling`** controls which of these quantities is requested from the QM interfaces and used in the propagation. The first option is **`nacdr`**, which requires the nonadiabatic coupling vectors $\langle\psi_\alpha|\partial/\partial\mathbf{R}|\psi_\beta\rangle$. For the wave function propagation, the scalar product of these vectors and the nuclear velocity is calculated to obtain the matrix $\langle\psi_\alpha|\partial/\partial t|\psi_\beta\rangle$. During the propagation, this matrix is interpolated linearly within each classical time step. Currently, only few SHARC interfaces can provide these couplings.

Alternatively, one can directly request the matrix elements $\langle\psi_\alpha|\partial/\partial t|\psi_\beta\rangle$, which can be used for the propagation. The corresponding argument to **`coupling`** is **`nacdt`**. In this case, the matrix is taken as constant throughout each classical time step. Currently, none of the interfaces in SHARC can deliver these couplings, because they are computed via overlaps, and if overlaps are known it is preferable to use local diabatization.

The third possibility is the use of the overlap matrix, requested with **`coupling overlaps`** (this is the default). The overlap matrix is used subsequently in the local diabatization algorithm for the wave function propagation. Currently, all SHARC interfaces can provide these couplings.

Correction to the Diagonal Gradients As detailed in ??, the correct transformation of the gradients to the diagonal representation includes contributions from the nonadiabatic coupling vectors. Using **`gradcorrect`**, these contributions are included. In this case SHARC will request the calculation of the nonadiabatic coupling vectors, even if they are not used in the wave function propagation. In order to explicitly turn off this gradient correction, use the **`nogradcorrect`** keyword.

Frustrated Hops and Adjustment of the Kinetic Energy The keyword **`ekincorrect`** controls how the kinetic energy is adjusted after a surface hop to preserve total energy. **`ekincorrect none`** deactivates the adjustment, so that the total energy is not preserved after a hop. Using this option, jumps can never be frustrated and are always performed according to the hopping probabilities. Using **`ekincorrect parallel_vel`**, the kinetic energy is adjusted by simply rescaling the nuclear velocities so that the new kinetic energy is $E_{\mathrm{tot}} - E_{\mathrm{pot}}$. Jumps are frustrated if the new potential energy would exceed the total energy. Using **`ekincorrect parallel_nac`**, the kinetic energy is adjusted by rescaling the component of the nuclear velocities parallel to the nonadiabatic coupling vector between the old and new state. The hop is frustrated if there is not enough kinetic energy in this direction to conserve total energy. Note that **`ekincorrect parallel_nac`** implies the calculation of the nonadiabatic coupling vector, even if they are not used for the wave function propagation. Finally, using **`ekincorrect parallel_diff`**, the kinetic energy is adjusted by rescaling the component of the nuclear velocities parallel to the difference gradient vector between the old and new state. The hop is frustrated if there is not enough kinetic energy in this direction to conserve total energy.

The keyword **`reflect_frustrated`** furthermore controls whether the velocities are inverted after a frustrated hop. With **`reflect_frustrated none`** (the default), after a frustrated hop, the velocity vector is not modified. Using **`reflect_frustrated parallel_vel`**, the full velocity vector is inverted when a frustrated hop is encountered. With the third option, **`reflect_frustrated parallel_nac`**, only the velocity component parallel to the nonadiabatic coupling vector between the active and frustrated states is inverted. This implies the calculation of the nonadiabatic coupling vector, even if they are not used for the wave function propagation. With **`reflect_frustrated parallel_diff`**, only the velocity component parallel to the gradient difference vector between the active and frustrated states is inverted.

Decoherence Correction Scheme There are three options for the decoherence correction (see ??) in SHARC, which can be selected with the `decoherence_scheme` keyword.

With the default `decoherence_scheme none`, no decoherence correction is applied. The energy-difference based decoherence (EDC) scheme of Granucci et al. [?] can be activated with `decoherence_scheme edc`. The keyword `decoherence_param` can be used to change the relevant parameter α (see ??). The default is 0.1 Hartree, which is the value recommended by Granucci et al. [?]. Alternatively, the AFSSH (augmented fewest-switches surface hopping) scheme of Jain et al. [?] can be employed. This scheme does not use any parameters, so the keyword `decoherence_param` will have no effect. Note that in any case, the decoherence correction is applied to the states in the representation chosen with the `surf` keyword.

The keywords `decoherence` (activates EDC decoherence) and `nodecoherence` are present for backwards compatibility.

Surface Hopping Scheme There are three options for the computation of the hopping probabilities (see ??) in SHARC, which can be selected with the `hopping_procedure` keyword.

Using `hopping_procedure off`, surface hopping will be disabled, such that the active state (in the representation chosen with the `surf` keyword) will never change. With the default, `hopping_procedure sharc`, the standard hopping probability equation from Ref. [?] will be used. Alternatively, one can use the global flux surface hopping scheme [?], which might be advantageous in super-exchange situations.

One can also turn off surface hopping with the `no_hops` keyword.

Forced Hops to Ground State With this option, one can force SHARC to hop from the active to the lowest state if the energy gap between these two states falls below a certain threshold. The threshold is given as argument to the `force_hop_to_gs` keyword (in eV). This option is useful for single-reference methods that fail to converge if the ground state-excited state energy gap becomes too small. c Note that this option forces hops from any higher-lying state to the lowest, independent whether there are other states between the active and the lowest state. Also note that once in the lowest state, hopping is completely forbidden if this option is active.

Atom Masking Some of the above surface hopping settings might not be fully size consistent: (i) in `ekincorrect parallel_vel`, all atoms are uniformly accelerated/slowed during velocity rescaling; (ii) in `reflect_frustrated parallel_vel`, the velocities of all atoms are inverted; (iii) with `decoherence_scheme edc`, the kinetic energy of all atoms determines the decoherence rate. In large systems (e.g., in solution), these effects might be unrealistic, because, e.g., a surface hop in the chromophore should not uniformly slow down all water molecules.

The `atommask` keyword can then be used to exclude certain atoms from the three mentioned procedures. With `atommask external`, the list of masked and active atoms is read from the file specified with the `atommaskfile` keyword (default `"atommask"`). The format of this file is described in section ??. With the other possible option, `atommask none` (the default), all atoms are considered for these procedures.

Note that the `atommask` keyword has no effect on `ekincorrect parallel_nac`, `reflect_frustrated parallel_nac`, and `decoherence_scheme afssh`, because these procedures are size consistent by themselves.

Reference Energy The keyword `ezero` gives the energy shift for the diagonal elements of the Hamiltonian. The shift should be chosen so that the shifted diagonal elements are reasonably small (large diagonal elements in the Hamiltonian lead to rapidly changing coefficients, requiring extremely short subtime steps).

Note that the energy shift default is 0, i.e., SHARC does not choose an energy shift based on the energies at the first time step (this would lead to each trajectory having a different energy shift).

Scaling and Damping The scaling factor for the energies and gradients must be positive (not zero), see section ??.

The damping factor must be in the interval [0, 1] (first, since the kinetic energy is always positive; second, because a damping factor larger than 1 would lead to exponentially growing kinetic energy). Also see section ??.

Selection of Gradients and Non-Adiabatic Couplings SHARC allows to selectively calculate only certain gradients and nonadiabatic coupling vectors at each time step. Those gradients and nonadiabatic coupling vectors not selected are not requested from the interfaces, thus decreasing the computational cost. The selection procedure is detailed in ??. Selection of gradients is activated by `grad_select`, selection for nonadiabatic couplings by `nac_select`. Selection is turned off by default.

The selection procedure picks only states which are closer in energy to the classically occupied state than a given threshold. The threshold is 0.5 eV by default and can be adjusted using the **eselect** keyword.

Since SHARC2.1, by default the **select_directly** keyword is active, which tells SHARC to use the energies of the last time step for selecting, so that only one call per time step is necessary. The alternative (keyword **noselect_directly**) is to perform two quantum chemistry calls per time step. In the first call, all quantities are requested except for the ones to be selected. The energies are used to determine which gradients and NACs to calculate in a second quantum chemistry call. The option **select_directly** is strongly recommended in almost all instances, since for most quantum chemistry programs it is not possible to make sure that the wave function phases from both calls are consistent. Additionally, for all interfaces the calculation becomes more expensive with two calls per step.

Phase Tracking Phase tracking is an important ingredient in SHARC. It is necessary for two reasons: (i) the columns of the transformation matrix **U** are determined only up to an arbitrary phase factor $e^{i\phi}$ (and additional mixing angles in case of degeneracy), and (ii) the wave functions produced by any quantum chemistry code are determined only up to an arbitrary sign. Both kind of phases need to be tracked in SHARC in order to obtain smoothly varying matrix elements which can be properly integrated.

By default, SHARC automatically tracks the phases in the **U** matrix (explicit keyword: **track_phase**), because all required information is always available. This phase tracking can be deactivated with the **notrack_phase** keyword, which can provide a significant speed advantage for **pysharc** calculations. However, you should check whether your results are affected by **notrack_phase**, and revert to **track_phase** if they do.

The tracking of the wave function signs depends on the interfaces, because only they have access to the explicit form of the wave functions. SHARC by default (explicit keyword: **phases_from_interface**) requests that the interface tracks the signs and reports any sign changes to SHARC. Currently, all interfaces can provide this phase information, but all of them need to perform overlap calculations to do so. The **nophases_from_interface** keyword can be used to deactivate these requests.

In some situations, it might be necessary to have consistent wave function signs between different trajectories. In this case, the **phases_at_zero** keyword can be used to compute sign information at $t = 0$; this requires that the relevant wave function data of the reference is already located in the **restart/** directory before the trajectory is started. Note that **phases_at_zero** is therefore an expert option.

Spin-Orbit Couplings Using the keyword **nospinorbit** the calculation of spin-orbit couplings is disabled. SHARC will only request the diagonal elements of the Hamiltonian from the interfaces. If the interface returns a non-diagonal Hamiltonian anyways, the off-diagonal elements are deleted.

The keyword **spinorbit** (which is the default) enables spin-orbit couplings.

Dipole Moment Gradients The derivatives of the dipole moments can be included in the gradients. This can be activated with the keyword **dipole_gradient**. Currently, only the analytical and MOLCAS interfaces can deliver these quantities.

Ionization The keyword **ionization** activates (**noionization** deactivates) the on-the-fly calculation of ionization transition properties. If the keyword is given, by default these properties are calculated every time step. The keyword **ionization_step** can be used to calculate these properties only every n-th time step. If the keyword is given, SHARC will request the calculation of the ionization properties from the interface, which needs to be able to calculate them.

The ionization probabilities are treated as one 2D property matrix, hence **n_property2d** should be at least 1.

THEODORE The keyword **theodore** activates (**notheodore** deactivates) the on-the-fly calculation of wave function descriptors with the THEODORE program. This can be very useful to track the wave function character of the states on-the-fly. The interface must be able to execute and THEODORE and return its output to SHARC (currently, the ADF, GAUSSIAN, TURBOMOLE, and ORCA interfaces can do this). The keyword **theodore_step** can be used to calculate these descriptors only every n-th time step.

The THEODORE descriptors are treated as one 1D property vector for each descriptor, and **n_property1d** should be at least as large as the number of descriptors computed by the interface.

Output control There are a number of keywords which control what information is written to the **output.dat** file. These keywords are **write_grad**, **write_nacdr**, **write_overlap**, **write_property1d**, and **write_property2d** (and the inverse of each one, e.g., **nowrite_grad**). Only **write_overlap** is activated by default, because it does not enlarge the data file by much, and contains important information which is read by **data_extractor.x**. **write_grad** and **write_nacdr** are turned off by default; they are primarily intended for users who want to keep all quantum chemical data, e.g., for training in machine learning. The keywords **write_property1d** and **write_property2d** are automatically activated if **theodore** or **ionization** (respectively) are activated.

Output writing stride The keyword **output_dat_steps** can be used to control how often data is written to **output.dat**. This is useful to reduce the amount of data generated by long trajectories (e.g., with the LVC or Analytical interfaces).

In the simplest version, using **output_dat_steps N** will set the output stride to **N**, so that **output.dat** is updated every **N**-th step (i.e., if step modulo **N** is equal to zero). The default is to write every step. Because in nonadiabatic dynamics, usually ballistic processes occur rapidly in the beginning and statistical processes occur slowly for longer times, this keyword allows printing more often in the beginning and less often for longer times. To use this feature, write **output_dat_steps N1 M2 N2**, which will use **N1** as the stride if step is larger or equal to zero, and **N2** as the stride if step is larger or equal to **M2**. One can also use three different strides via **output_dat_steps N1 M2 N2 M3 N3**, but not more than three.

Note that for these numbers **sharc.x** enforces **N**≥1 and **M**≥0, but no particular ordering. Hence, it is possible to use longer strides in the beginning and smaller strides later, although this is rarely useful. If step is larger/equal to both **M2** and **M3**, then **N3** will be used.

Also note that the data written to **output.dat** is always simply the data for the respective time step, no average over the last **N** steps. This needs to be kept in mind when analyzing the trajectory plots.

Output format See sections ?? and ?? for details.

4.1.4 Example

The following input sample shows a typical input for excited-state dynamics including IC within a singlet manifold plus intersystem crossing to triplet states. It includes a large number of excited singlet states in order to calculate transient absorption spectra. Only the lowest three singlet states actually participate in the dynamics.

```
nstates   8 0 3       # many singlet states for transient absorption
actstates 3 0 3       # only few states to reduce gradient costs

stepsize 0.5          # typical time step for a molecule containing H
tmax 1000.0           # one picosecond

surf diagonal
state 3 mch                  # start on the S2 singlet state
coeff auto                   # coefficient of S2 will be set to one
coupling overlap             # \
decoherence_scheme edc       # | typical settings
ekincorrect parallel_vel     # |
gradcorrect                  # /
grad_select           # \
nac_select            # | improve performance
eselect 0.3           # /

veloc external        # velocities come from file "veloc"
velocfile "veloc"     #

RNGseed 65435
ezero -399.41494751   # ground state energy of molecule
```

4.2 Geometry file

The geometry file (default file name is **geom**) contains the initial coordinates of all atoms. This file must be present when starting a new trajectory.

The format is based on the COLUMBUS geometry file formathttp://www.univie.ac.at/columbus/docs$_C OL70/documentation_m ain.htmlC$o (however, SHARC is more flexible with the formatting of the numbers). For each atom, the file contains one line, giving the chemical symbol (a string), the atomic number (a real number), the x, y and z coordinates of the atom in Bohrs (three real numbers), and the relative atomic weight of the atom (a real number). The six items must be separated by spaces. The real numbers are read in using Fortran list-directed I/O, and hence are free format (can have any numbers of decimals, exponential notation, etc.). Element symbols can have at most 2 characters.

The following is an example of a **geom** file for CH_2:

```
C 6.0  0.0 0.0  0.0 12.000
H 1.0  1.7 0.0 -1.2  1.008
H 1.0  1.7 0.0  3.7  1.008
```

4.3 Velocity file

The velocity file (default **veloc**) contains the initial nuclear velocities (e.g., from a Wigner distribution sampling). This file is optional (the velocities can be initialized with the **veloc** input keyword).

The file contains one line of input for each atom, where the order of atoms must be the same as in the **geom** file. Each line consists of three items, separated by spaces, where the first is the x component of the nuclear velocity, followed by the y and z components (three real numbers). The input is interpreted in atomic units (Bohr/atu).

The following is an example of a **veloc** file:

```
 0.0001  0.0000  0.0002
 0.0002  0.0000  0.0012
 0.0003  0.0000 -0.0007
```

4.4 Coefficient file

The coefficient file contains the initial wave function coefficients. The file contains one line per state (total number of states, i.e., multiplets count multiple times). Each line specifies the initial coefficient of one state. If the initial state is specified in the MCH representation (input keyword **state**), then the order of the initial coefficients must be as given by the canonical ordering (see section **??**). If the initial state is given in diagonal representation, then the initial coefficients correspond to the states given in energetic ordering, starting with the lowest state. Each line contains two real numbers, giving first the real and then the imaginary part of the initial coefficient of the respective state. Note that after read-in, the coefficient vector is normalized to one.

Example:

```
0.0 0.0
1.0 0.0
0.0 0.0
```

4.5 Laser file

The laser file contains a table with the amplitude of the laser field $\boldsymbol{\epsilon}(t)$ at each time step of the *electronic* propagation. Given a laser field of the general form:

$$\boldsymbol{\epsilon}(t) = \begin{pmatrix} \Re(\epsilon_x(t)) + \mathrm{i}\Im(\epsilon_x(t)) \\ \Re(\epsilon_y(t)) + \mathrm{i}\Im(\epsilon_y(t)) \\ \Re(\epsilon_z(t)) + \mathrm{i}\Im(\epsilon_z(t)) \end{pmatrix} \tag{4.1}$$

each line consists of 8 elements: t (in fs), $\Re(\epsilon_x(t))$, $\Im(\epsilon_x(t))$, $\Re(\epsilon_y(t))$, $\Im(\epsilon_y(t))$, $\Re(\epsilon_z(t))$, $\Im(\epsilon_z(t))$, (all in atomic units), and finally the instantaneous central frequency (also atomic units).

The time step in the laser file must exactly match the time step used for the electronic propagation, which is the time step used for the nuclear propagation (keyword **`stepsize`**) divided by the number of substeps (keyword **`nsubsteps`**). The first line of the laser file must correspond to t=0 fs.

Example:

```
0.00E+00 -0.68E-03  0.00E+00  0.00E+00  0.00E+00  0.00E+00  0.00E+00  0.31E+00
0.10E-02 -0.77E-02  0.00E+00  0.00E+00  0.00E+00  0.00E+00  0.00E+00  0.33E+00
0.20E-02 -0.13E-01  0.00E+00  0.00E+00  0.00E+00  0.00E+00  0.00E+00  0.35E+00
     ...       ...       ...       ...       ...       ...       ...       ...
```

4.6 Atom mask file

The atom mask file contains for each atom a line with a Boolean entry ("T" or "F"), which indicates whether the atom should be considered in the relevant procedures. Specifically, the atom masking settings affect the options **`ekincorrect parallel_vel`**, **`reflect_frustrated parallel_vel`**, and **`decoherence_scheme edc`**. In all cases, "T" indicates that the atom should be considered (as if **`atommask`** was not given), whereas "F" indicates that the atom should be ignored for these procedures.

Example:

```
 T
 T
 F
 F
...
```

5 Output files

This chapter documents the content of the output files of SHARC. Those output files are **output.log**, **output.lis**, **output.dat** and **output.xyz**.

5.1 Log file

The log file **output.log** contains general information about all steps of the SHARC simulation, e.g., about the parsing of the input files, results of quantum chemistry calls, internal matrices and vectors, etc. The content of the log file can be controlled with the keyword **printlevel** in the SHARC main input file.

In the following, all printlevels are explained.

Printlevel 0 At printlevel 0, only the execution infos (date, host and working directory at execution start) and build infos (compiler, compile date, building host and working directory) are given.

Printlevel 1 At printlevel 1, also the content of the input file (cleaned of comments and blank lines) is echoed in the log file. Also, the start of each time step is given.

Printlevel 2 At printlevel 2, the log file also contains information about the parsing of the input files (echoing all enabled options, initial geometry, velocity and coefficients, etc.) and about the initialization of the coefficients after the first quantum chemistry calculation. This printlevel is recommended for production calculations, since it is the highest printlevel where no output per time step is written to the log file.

Printlevel 3 This and higher printlevels add output per time step to the log file. At printlevel 3, the log file contains at each time step the data from the velocity-Verlet algorithm (old and new acceleration, velocity and geometry), the old and new coefficients, the surface hopping probabilities and random number, the occupancies before and after decoherence correction as well as the kinetic, potential and total energies.

Printlevel 4 At printlevel 4, additionally the log file contains information on the quantum chemistry calls (file names, which quantities were read, gradient and nonadiabatic coupling vector selection) and the propagator matrix.

Printlevel 5 At printlevel 5, additionally the log file contains the results of each quantum chemistry calls (all matrices and vectors), all matrices involved in the propagation as well as the matrices involved in the gradient transformation. This is the highest printlevel currently implemented.

5.2 Listing file

The listing file **output.lis** is a tabular summary of the progress of the dynamics simulation. At the end of each time step (including the initial time step), one line with 11 elements is printed. These are, from left to right:

1. current step (counting starts at zero for the initial step),

2. current simulation time (fs),
3. current state in the diagonal representation,
4. approximate corresponding MCH state (see subsection ??),
5. kinetic energy (eV),
6. potential energy (eV),
7. total energy (eV),
8. current gradient norm (in eV/Å),
9. current expectation values of the state dipole moment (Debye),
10. current expectation values of total spin,
11. wallclock time needed for the time step.

The listing file also contains one extra line for each surface hopping event. For accepted hops, the old and new states (in diagonal representation) and the random number are given. Frustrated hops and resonant hops are also mentioned. Note that the extra line for surface hopping occurs before the regular line for the time step.

The listing file can be plotted with standard tools like GNUPLOT and can be read with **`data_collector.py`**.

Energies The kinetic energy is calculated at the end of each time step (i.e., after surface hopping events and the corresponding adjustments). The potential energy is the energy of the currently active diagonal state. The total energy is the sum of those two.

Expectation values The gradient norms given in the listing file is calculated as follows:

$$g_{\text{list}} = \sqrt{\frac{1}{3N_{\text{atom}}}\sum_{a}^{N_{\text{atom}}}\sum_{d=x,y,z} g_{ad}^2} \tag{5.1}$$

which is then transformed to eV/Å.

The expectation values of the dipole moment for the active state β is calculated from:

$$\mu = \sqrt{\sum_{p=x,y,z}\left(\sum_{\sigma}\sum_{\tau}\Re\left[U_{\beta\sigma}^{\dagger}\mu_{\sigma\tau}^{p}U_{\tau\beta}\right]\right)^2} \tag{5.2}$$

The expectation value of the total spin of the active state β is calculated as follows:

$$S = \sum_{\alpha}|U_{\alpha\beta}|^2 S_{\alpha} \tag{5.3}$$

where S_α is the total spin of the MCH state with index α.

5.3 Data file

The data file **`output.dat`** contains all relevant data from the simulation for all time steps, in ASCII format. Accordingly, this file can become quite large for long trajectories or if many states are included, but for most file systems it is easier to deal with a single large file than with many small files.

Usually, after the simulation is finished the data file is processed by **`data_extractor.x`** to obtain a number of tabular files which can be plotted or post-processed (e.g., with **`data_collector.py`**). For this, see sections ?? for the data extractor, ?? for plotting, and ?? for post-processing.

5.3.1 Specification of the data file

The data file format was changed from the first release version of SHARC. The new format uses a different header, which is keyword-based (like the **`input`** file) and starts with **`SHARC_version 2.0`**. The general structure of the time step data is the same as in the first release version.

The data file contains a short header followed by the data per time step. All quantities are commented in the data file. The header is keyword-based and contains at least the following entries:

1. number of states per multiplicity,
2. number of atoms,
3. number of 1D properties,
4. number of 2D properties,
5. time step,
6. **write_overlap**,
7. **write_grad**,
8. **write_nacdr**,
9. **write_property1d**,
10. **write_property2d**,
11. information whether a laser field is included.

At the end of the header, the data file contains a header array section. Currently, this includes:

1. atomic numbers,
2. elements,
3. masses,
4. full laser field for all substeps (only if flag is set).

The entry for each time step contains:

1. step index
2. Hamiltonian in MCH representation,
3. transformation matrix **U**,
4. MCH dipole moment matrices (x, y, z),
5. overlap matrix in MCH representation (only if flag is set),
6. coefficients in the diagonal representation,
7. hopping probablities in the diagonal representation
8. kinetic energy,
9. currently active state in diagonal representation and approximate state in MCH representation,
10. random number for surface hopping,
11. wallclock time (in seconds)
12. geometry (Bohrs),
13. velocities (atomic units),
14. 2D property matrices (only if flag is set),
15. 1D property vectors (only if flag is set),
16. gradient vectors (only if flag is set),
17. nonadiabatic coupling vectors (only if flag is set).

5.4 Data file in NetCDF format

If **output_format** is set to **netcdf**, then only the header will be written to the **output.dat** file, but no information per time step. The latter is instead written in (binary) NetCDF format to **output.dat.nc**.

This option is intended to provide faster output for fast simulations with **pysharc**. However, it is also an option that leads to less disk memory consumption, and can be used with **sharc.x**. To this end, **output_format**=**netcdf** also deactivates writing of the **output.xyz** file. Also, when NetCDF files are written, no restart files are produced, except on the last time step or in case of errors.

NetCDF output files can be extracted with **data_extractor_NetCDF.x** (see section ??). This program can also generate the **output.xyz** for further analysis.

Currently, there are a few limitations in this approach. Most importantly, the NetCDF files do not store the property vectors and property matrices that can be stored in the **output.dat** files. Hence, computations with the **ionization** or **theodore** keywords should not use the NetCDF format.

The program **data_converter.x** (see section ??) can be used to convert a **output.dat** file into a NetCDF file. This can be useful when archiving an ensemble of trajectories, as the NetCDF files are much smaller than the ASCII files. Note that after conversion, the **output.xyz** files can be deleted, as they can be regenerated from the NetCDF file.

5.5 XYZ file

The file **output.xyz** contains the geometries of all time steps in standard xyz file format. It can be used with visualization programs like MOLDEN, GABEDIT or MOLEKEL to create movies of the molecular motion, or with **geo.py** (see ??) to calculate internal coordinates for each time step. Furthermore, **trajana_nma.py** (see ??) and **trajana_essdyn.py** (see ??) read this file.

The comments of the geometries (given in the second line of each geometry block) contain information about the simulation time and the active state (first in diagonal basis, then in MCH basis).

If **output_format=netcdf**, the **output.xyz** file will be empty. Use **data_extractor_NetCDF.x** with the **-xyz** flag to obtain the **output.xyz** file.

6 Interfaces

This chapter describes the interface between SHARC and quantum chemistry programs. In the first section, the interface is specified (e.g., for users who attempt to create their own interfaces). The description of the currently existing interfaces takes the remainder of this chapter.

6.1 Interface Specifications

From the SHARC point of view, quantum chemical calculation proceeds as follows in the **`QM`** directory:

1. write a file called **`QM/QM.in`**
2. call a script called **`QM/runQM.sh`**
3. read the output from a file called **`QM/QM.out`**

For specifications of the formats of these two files (**`QM.in`** and **`QM.out`**) see below. The executable script **`QM/runQM.sh`** must accomplish that all necessary quantum chemical output is available in **`QM/QM.out`**.

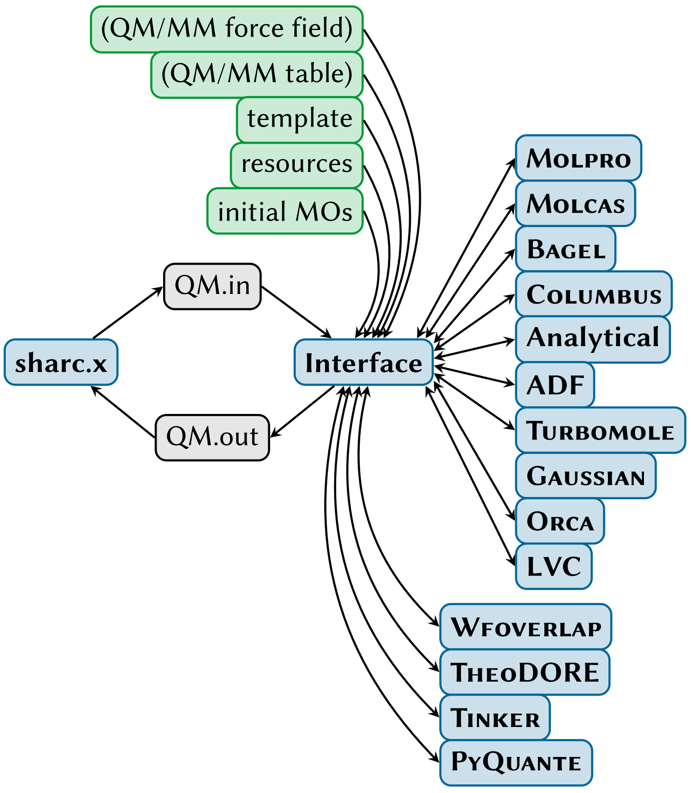

Figure 6.1: Communication between **`sharc.x`**, the interfaces, and the quantum chemistry codes.

6.1.1 `QM.in` Specification

The **`QM.in`** file is written by SHARC every time a quantum chemistry calculation is necessary. It contains all information available to SHARC. This information includes the current geometry (and velocity), the time step, the number of states, the time step and the unit used to specify the atomic coordinates. The file also contains *control* keywords and *request* keywords.

The file format is consistent with a standard xyz file. The first line contains the number of atoms, the second line is a comment. SHARC writes the trajectory ID (a hash of all SHARC input files) to this line. The following lines specify the atom positions. As a fourth, fifth and sixth column, these lines may contain the atomic velocities. All following lines contain keywords, one per line and possibly with arguments. Comments can be inserted with '#', and empty lines are permitted. Comments and empty lines are only permitted below the xyz file part. An examplary **`QM.in`** file is given in the following:

```
3
Jobname
S     0.0     0.0     0.0    0.000  -0.020   0.002
H     0.0     0.9     1.2    0.000  -0.030   0.000
H     0.0    -0.9     1.2    0.000   0.010  -0.000
# This is a comment
Init
States 3 0 2
Unit Angstrom
SOC
DM
GRAD 1 2
OVERLAP
NACDR select
  1 2
  end
```

There exist two types of keywords, *control* keywords and *request* keywords. Control keywords pass some information to the interface. Request keywords tell the interface to provide a quantity in the **`QM.out`** file. Table **??** contains all control keywords while table **??** lists all request keywords.

6.1.2 `QM.out` Specification

The **`QM.out`** file communicates back the results of the quantum chemistry calculation to the dynamics code. After SHARC called **`QM/runQM.sh`**, it expects that the file **`QM/QM.out`** exists and contains the relevant data.

The following quantities are expected in the file (depending whether the corresponding keyword is in the **`QM.in`** file): Hamiltonian matrix, dipole matrices, gradients, nonadiabatic couplings (either NACDR or NACDT), overlaps, wave function phases, property matrices. The format of **`QM.out`** is described in the following.

Each quantity is given as a data block, which has a fixed format. The order of the blocks is arbitrary, and between blocks arbitrary lines can be written. However, within a block no extraneous lines are allowed. Each data block starts with a exclamation mark **`!`**, followed by whitespace and an integer flag which specifies the type of data:

Table 6.1: Control keywords for SHARC interfaces. These keywords pass information from SHARC to the interface.

Keyword	Description
`init`	Specifies that this is the first calculation. The interface should create a save directory (if not existing already) to save all information necessary for a restart.
`samestep`	Specifies that this is an additional calculation at the same geometry/time step. Implies that, e.g., the converged orbitals from the savedir could be reused or that the wave function calculations could be skipped.
`restart`	Specifies that this is a calculation after a restart of **`sharc.x`**, possibly after a crash. Implies that in the savedir some files might be incomplete, the interface should therefore act as if a new step is requested, except that the savedir files should be managed accordingly.
`cleanup`	Specifies that all output files of the interface (except **`QM.out`**) should be deleted (including the save directory).
`backup`	Specifies that the content of the save directory should be stored in a persistent location (such that it is not overwritten in the next time step).
`unit`	Specifies in which unit the atomic coordinates are to be interpreted. Possible arguments are “angstrom” and “bohr”.
`states`	Gives the number of excited states per multiplicity (singlets, doublet, triplets, ...).
`dt`	Gives the time between the last calculation and the current calculation in atomic units.
`savedir`	Gives a path to the directory where the interface should save files needed for restart and between time steps. If the interface-specific input files also have this keyword, SHARC assumes that the path in **`QM.in`** takes precedence.

Table 6.2: Request keywords for SHARC interfaces. See Table **??** for which interfaces can fulfill these requests.

Keyword	Description
`H`	Calculate the molecular Hamiltonian (diagonal matrix with the energies of the states of the model space). This request is always available.
`SOC`	Calculate the molecular Hamiltonian including the SOCs (not diagonal anymore within the model space).
`DM`	Calculate the state dipole moments and transition dipole moments between all states.
`GRAD`	Calculate gradients for all states. If followed by a list of states, calculate only gradients for the specified states.
`NACDT`	Calculate the time-derivatives $\langle\Psi_1\vert\partial/\partial t\vert\Psi_2\rangle$ by finite differences between the last time step and the current time step. This request is currently not supported by any interface.
`NACDR`	Calculate nonadiabatic coupling vectors $\langle\Psi_1\vert\partial/\partial\mathbf{R}\vert\Psi_2\rangle$ between all pairs of states. If followed by “select”, read the list of pairs on the following lines until “end” and calculate nonadiabatic coupling vectors between the specified pairs of states.
`OVERLAP`	Calculate overlaps $\langle\Psi_1(t_0)\vert\Psi_2(t)\rangle$ between all pairs of states (between the last and current time step). If followed by “select”, read the list of pairs on the following lines until “end” and calculate overlaps between the specified pairs of states.
`DMDR`	Calculate the Cartesian gradients of the dipole moments and transition dipole moments of all states.
`ION`	Calculate transition properties between neutral and ionic wave functions.
`THEODORE`	Run THEODORE to compute electronic descriptors for all states.
`MOLDEN`	Generate MOLDEN files of the relevant orbitals and copy them to the savedir.

1	Hamiltonian matrix
2	Dipole matrices
3	Gradients
4	Non-adiabatic couplings (NACDT)
5	Non-adiabatic couplings (NACDR)
6	Overlap matrix
7	Wavefunction phases
8	Wallclock time for QM calculation
11	Property matrix (e.g., ionization probabilities) **this flag is deprecated**
12	Dipole moment gradients
20	Property matrices with number and labels (e.g., ionization probabilities)
21	Property vectors with number and labels (e.g., THEODORE output)

On the next line, two integers are expected giving the dimensions of the following matrix. Note, that all these matrices must be square matrices. On the following lines, the matrix or vector follows. Matrices are in general complex, and real and imaginary part of each element is given as a pair of floating point numbers.

The following shows an example of a 4×4 Hamiltonian matrix. Note that the imaginary parts directly follow the real parts (in this example, the Hamiltonian is real).

```
 ! 1
 4 4
-548.6488 0.0000    0.0000 0.0000    0.0003 0.0000    0.0003 0.0000
   0.0000 0.0000 -548.6170 0.0000    0.0003 0.0000    0.0003 0.0000
   0.0003 0.0000    0.0003 0.0000 -548.5986 0.0000    0.0000 0.0000
   0.0003 0.0000    0.0003 0.0000    0.0000 0.0000 -548.5912 0.0000
```

The three dipole moment matrices (x, y and z polarization) must follow directly after each other, where the dimension specifier must be present for each matrix. The dipole matrices are also expected to be complex-valued.

```
 ! 2
 2 2
 0.1320 0.0000 -0.0020 0.0000
-0.0020 0.0000 -1.1412 0.0000
 2 2
 0.0000 0.0000  0.0000 0.0000
 0.0000 0.0000  0.0000 0.0000
 2 2
 2.1828 0.0000  0.0000 0.0000
 0.0000 0.0000  0.6422 0.0000
```

Gradient and nonadiabatic couplings vectors are written as $3 \times n_{\text{atom}}$ matrices, with the x, y and z components of one atom per line. These vectors are expected to be real valued. Each vector is preceeded by its dimensions.

```
6 3
 0.0000 -6.5429 -8.1187
 0.0000  5.8586  8.0160
 0.0000  6.8428  1.0265
 0.0000  6.5429  8.1187
 0.0000 -5.8586 -8.0160
 0.0000 -6.8428 -1.0265
```

If gradients are requested, SHARC expects every gradient to be present, even if only some gradients are requested. The gradients are expected in the canonical ordering (see section **??**), which implies that for higher multiplets the same gradient has to be present several times. For example, with 3 singlets and 3 triplets, SHARC expects 12 gradients in the **`QM.out`** file (each triplet has three components with M_s = -1, 0 or 1).

Similarly, for nonadiabatic coupling vectors, SHARC expects all pairs, even between states of different multiplicity. The vectors are also in canonical ordering, where the inner loop goes over the ket states. For example, with 3 singlets and 3 triplets (12 states), SHARC expects 144 (12^2) nonadiabatic coupling vectors in the **`QM.out`** file.

```
! 5 Non-adiabatic couplings (ddr) (2x2x1x3, real)
1 3 ! m1 1 s1 1 ms1 0   m2 1 s2 1 ms2 0
 0.0e+0  0.0e+0  0.0e+0
1 3 ! m1 1 s1 1 ms1 0   m2 1 s2 2 ms2 0
+2.0e+0  0.0e+0  0.0e+0
1 3 ! m1 1 s1 2 ms1 0   m2 1 s2 1 ms2 0
-2.0e+0  0.0e+0  0.0e+0
1 3 ! m1 1 s1 2 ms1 0   m2 1 s2 2 ms2 0
 0.0e+0  0.0e+0  0.0e+0
```

The nonadiabatic coupling matrix (NACDT keyword), the overlap matrix and the property matrix are single $n \times n$ matrices (n is the total number of states), respectively, like the Hamiltonian.

The wave function phases are a vector of complex numbers.

The wallclock time is a single real number.

The dipole moment gradients are a list of $3 \times n_{\text{atom}}$ vectors, each specifying the gradient of one polarization of one dipole moment matrix element. In the outmost loop, the bra index is counted, then the ket index, then the polarization. Hence, the respective entry in **`QM.out`** would look like (for 2 states and 1 atom):

```
! 12 Dipole moment derivatives (2x2x3x1x3, real)
1 3 ! m1 1 s1 1 ms1 0   m2 1 s2 1 ms2 0   pol 0
 0.000000000000E+00  0.000000000000E+00  0.000000000000E+00
1 3 ! m1 1 s1 1 ms10   m2 1 s2 1 ms2 0  pol 1
 0.000000000000E+00  0.000000000000E+00  0.000000000000E+00
1 3 ! m1 1 s1 1 ms10   m2 1 s2 1 ms2 0  pol 2
 0.000000000000E+00  0.000000000000E+00  0.000000000000E+00
1 3 ! m1 1 s1 1 ms10   m2 1 s2 2 ms2 0  pol 0
 1.000000000000E+00  0.000000000000E+00  0.000000000000E+00
1 3 ! m1 1 s1 1 ms10   m2 1 s2 2 ms2 0  pol 1
 0.000000000000E+00  0.000000000000E+00  0.000000000000E+00
1 3 ! m1 1 s1 1 ms10   m2 1 s2 2 ms2 0  pol 2
 0.000000000000E+00  0.000000000000E+00  0.000000000000E+00
1 3 ! m1 1 s1 2 ms10   m2 1 s2 1 ms2 0  pol 0
 1.000000000000E+00  0.000000000000E+00  0.000000000000E+00
1 3 ! m1 1 s1 2 ms10   m2 1 s2 1 ms2 0  pol 1
 0.000000000000E+00  0.000000000000E+00  0.000000000000E+00
1 3 ! m1 1 s1 2 ms10   m2 1 s2 1 ms2 0  pol 2
 0.000000000000E+00  0.000000000000E+00  0.000000000000E+00
1 3 ! m1 1 s1 2 ms10   m2 1 s2 2 ms2 0  pol 0
 0.000000000000E+00  0.000000000000E+00  0.000000000000E+00
1 3 ! m1 1 s1 2 ms10   m2 1 s2 2 ms2 0  pol 1
 0.000000000000E+00  0.000000000000E+00  0.000000000000E+00
1 3 ! m1 1 s1 2 ms10   m2 1 s2 2 ms2 0  pol 2
 0.000000000000E+00  0.000000000000E+00  0.000000000000E+00
```

The section containing the 2D property matrices consists of three subsequent parts: (i) the number of property matrices contained, (ii) a label for each property matrix (as the property matrices might contain arbitrary data, depending on the interface and the requests), and (iii) the matrices (full, complex-valued matrices like above):

```
! 20 Property Matrices
2    ! number of property matrices
! Property Matrix Labels (1 strings)
```

```
Dyson norms
Example matrix
! Property Matrices (1x4x4, complex)
4 4   ! Dyson norms
 0.000E+00  0.000E+00  0.000E+00  0.000E+00  9.663E-01  0.000E+00  9.663E-01  0.000E+00
 0.000E+00  0.000E+00  0.000E+00  0.000E+00  4.822E-01  0.000E+00  4.822E-01  0.000E+00
 9.663E-01  0.000E+00  4.822E-01  0.000E+00  0.000E+00  0.000E+00  0.000E+00  0.000E+00
 9.663E-01  0.000E+00  4.822E-01  0.000E+00  0.000E+00  0.000E+00  0.000E+00  0.000E+00
4 4   ! Example matrix
 1.000E+00  1.000E+00  0.000E+00  0.000E+00  0.000E+00  0.000E+00  0.000E+00  0.000E+00
 0.000E+00  0.000E+00  2.000E+00  2.000E+00  0.000E+00  0.000E+00  0.000E+00  0.000E+00
 0.000E+00  0.000E+00  0.000E+00  0.000E+00  3.000E+00  3.000E+00  0.000E+00  0.000E+00
 0.000E+00  0.000E+00  0.000E+00  0.000E+00  0.000E+00  0.000E+00  4.000E+00  4.000E+00
```

The section containing the 1D property vectors also consists of three subsequent parts: (i) the number of property vectors contained, (ii) a label for each property vector (as the property vectors might contain arbitrary data, depending on the interface and the requests), and (iii) the vectors (real-valued):

```
! 21 Property Vectors
2    ! number of property vectors
! Property Vector Labels (2 strings)
Om
PRNTO
! Property Vectors (9x8, real)
! TheoDORE descriptor 1 (Om)
 0.000000000000E+000
 4.318700000000E-001
 2.688600000000E-001
 2.590000000000E-002
! TheoDORE descriptor 2 (PRNTO)
 0.000000000000E+000
 2.318700000000E-001
 1.688600000000E-001
 1.590000000000E-002
```

6.1.3 Further Specifications

The interfaces may require additional input files beyond **QM.in**, which contain static information. This may include paths to the quantum chemistry executable, paths to scratch directories, or input templates for the quantum chemistry calculation (e.g. active space specifications, basis sets, etc.). The dynamics code does not depend on these additional files, but they should all be stored in the **QM/** subdirectory.

The current conventions in the SHARC suite are that the quantum chemistry interfaces use two additional input files, one specifying the level of theory (template file, e.g., **MOLCAS.template**, **MOLPRO.template**, ...) and one specifying the computational resources like paths, memory, number of CPU cores, initial orbital source (resource file, e.g., **MOLCAS.resources**, **MOLPRO.resources**, ...). Furthermore, the current interfaces allow to read in initial orbitals (e.g., **MOLCAS.*.RasOrb.init**, **mocoef.init**, ...). For interfaces with QM/MM capabilities, additional files could be used to specify connection table, parameters, etc.

6.1.4 Save Directory Specification

The interfaces must be able to save all information necessary for restart to a given directory. The absolute path is written to **QM.in** by SHARC. Hence, for the trajectories the path to the save directory is always a subdirectory of the working directory of SHARC.

6.2 Overview over Interfaces

The SHARC suite comes with a number of interfaces to different quantum chemistry programs. Their capabilities and usage are explained in the following sections. Table **??** gives an overview over the capabilities of the interfaces. Table **??** shows the file names for interface-related input files of the different interfaces.

Table 6.3: Overview over capabilities of SHARC interfaces. For each method and program, the table shows which multiplicities (S^2), which quantities, and whether QM/MM are available.

Method	Program	S^2	SOC	TDM^a	Grad.	NACDR	OVL^b	DMDR	ION	Theo.^{c}	QM/MM
SA-CASSCF	MOLPRO	any	√	√	√	√	√		√		
	MOLCAS	any	√	√	√		√	√^{d}	√		√
	COLUMBUS	any	√^{e}	√	√	√^{e}	√		√		
	BAGEL	any		√	√	√	√		√		
MR-CISD	COLUMBUS	any	√^{e}	√	√	√^{e}	√		√		
MS-CASPT2	MOLCAS	any	√	√	√^{d}		√	√^{d}	√		
XMS-CASPT2	BAGEL	any		√	√	√	√		√		
TD-DFT	ADF	any	√^{f}	√^{g}	√		√		√	√	√
	GAUSSIAN	any		√^{g}	√		√		√	√	
	ORCA	any	√^{f}	√^{g}	√		√		√	√	√
ADC2	TURBOMOLE	S, T	√^{f}	√	√		√			√	√
CC2	TURBOMOLE	S, T		√^{g}	√		√			√	√
Analytical	–	any	√	√	√		√	√			
LVC	–	any	√	√	√	√	√				

^{a} TDM: transition dipole moments; ^{b} OVL: wave function overlaps; ^{c} Theo.: THEODORE; ^{d} numerical gradients; ^{e} either NAC or SOC, but not both at the same time; ^{f} SOCs only between singlets and triplets; ^{g} TDMs only between S_0 and excited singlets.

Table 6.4: Overview over files of SHARC interfaces.

Interface	Template file	Resource file	Initial MOs	QM/MM
MOLPRO	**`MOLPRO.template`**	**`MOLPRO.resources`**	**`wf.init`**	
			`wf.<job>.init`	
MOLCAS	**`MOLCAS.template`**	**`MOLCAS.resources`**	**`MOLCAS.<mult>.RasOrb.init`**	**`MOLCAS.qmmm.table`**
			`MOLCAS.<mult>.JobIph.init`	**`MOLCAS.qmmm.key`**
COLUMBUS	a directory	**`COLUMBUS.resources`**	**`mocoef_mc.init`**	
			`mocoef_mc.init.<job>`	
			`molcas.RasOrb.init`	
			`molcas.RasOrb.init.<job>`	
BAGEL	**`BAGEL.template`**	**`BAGEL.resources`**	**`archive.<mult>.init`**	
ADF	**`ADF.template`**	**`ADF.resources`**	**`ADF.t21.<job>.init`**	**`ADF.qmmm.table`**
				`ADF.qmmm.ff`
GAUSSIAN	**`GAUSSIAN.template`**	**`GAUSSIAN.resources`**	**`GAUSSIAN.chk.init`**	
			`GAUSSIAN.chk.<job>.init`	
ORCA	**`ORCA.template`**	**`ORCA.resources`**	**`ORCA.gbw.init`**	**`ORCA.qmmm.table`**
			`ORCA.gbw.<job>.init`	**`ORCA.qmmm.ff`**
TURBOMOLE	**`RICC2.template`**	**`RICC2.resources`**	**`mos.init`**	**`RICC2.qmmm.table`**
				`RICC2.qmmm.ff`
Analytical	**`Analytical.template`**	N/A	N/A	N/A
LVC	**`LVC.template`**	N/A	N/A	N/A

6.2.1 Example Directory

The directory **`$SHARC/../examples/`** contains for all quantum chemistry interfaces comprehensively commented examples of input files (template, resource files). These example files should be regarded as supplementary files to the documentation of the interfaces. For the interfaces without the possibility for automated template generation (e.g., **SHARC_RICC2.py**), it is recommended that users copy the example template file and modify it to their needs.

However, note that it might not necessarily work to directly start the respective interface in the example directories. In order to make the example calculations work, some paths or variables in the resource files need to be adjusted. If you need automatically working test calculations for SHARC, consider using **tests.py** instead.

6.3 MOLPRO Interface

The SHARC-MOLPRO interface allows to run SHARC dynamics with MOLPRO's CASSCF wave functions. RASSCF is not supported, since on RASSCF level state-averaging over different multiplicities is not possible. The interface uses MOLPRO's CI program in order to calculate transition dipole moments and spin-orbit couplings. Gradients and nonadiabatic coupling vectors are calculated using MOLPRO's CADPACK code (hence no generally contracted basis sets can be used). In the new version of the interface, overlaps are calculated using the WFOVERLAP code. Time-derivative couplings (NACDT) are not supported anymore in the SHARC-MOLPRO interface. Wavefunction phases between the CASSCF and MRCI wave functions are automatically adjusted. The interface can trivially parallelize the computation of gradients and coupling vectors over several processors. Execution of parallel MOLPRO binaries is currently not supported.

Important note: It appears that in some cases the sign of the nonadiabatic coupling vectors randomly changes along a trajectory ran with MOLPRO, and that it is not possible to identify this sign change from the MO and CI coefficients. Hence, propagation with **coupling nacdr** is currently discouraged for the SHARC-MOLPRO interface. Alternative possibilities to run SHARC-CASSCF dynamics with nonadiabatic coupling vectors are given by the MOLCAS (section **??**) and BAGEL (section **??**) interfaces.

The SHARC-MOLPRO interface needs two additional input files, which should be present in **QM/**. Those input files are **MOLPRO.resources**, which contains, e.g., the paths to MOLPRO and the scratch directory, and **MOLPRO.template**, which is a keyword-argument input file specifying the CASSCF level of theory. If **QM/wf.init** is present, it will be used as a MOLPRO wave function file containing the initial MOs. For calculations with several "jobs" (see below), initial orbitals can also given as **QM/wf.<job>.init**.

6.3.1 Template file: MOLPRO.template

During the rework of the MOLPRO interface, the template file structure was completely changed. In the new version, the template file is a keyword-argument list file, similar to the template files of most other interfaces. A fully commented template file with all possible options is located in **$SHARC/../examples/SHARC_MOLPRO/**.

For simple cases, an example for the template file looks like this:

```
basis def2-svp
dkho 2                  # Douglas-Kroll second order
occ 14
closed 10
nelec           24
roots           4 0 3
rootpad         1 0 1
```

This specifies a SA(4S+3T)-CASSCF(4,4)/def2-SVP calculation for 24 electrons. Note the **rootpad** keyword, which adds one singlet and one triplet with zero weight to the state-averaging (so technically this is a SA(5S+4T) calculation, but the results are the same as SA(4S+3T)). These zero-weight states are sometimes useful to improve convergence of CASSCF.

The new interface can also be used to perform several independent CASSCF calculations for different multiplicities (e.g., one CASSCF for the neutral states and another one for the ionic states). In this case, each independent CASSCF calculation is called a "job". In the template, most settings can be modified independently for each job. An example is given here:

```
# In this way, users can employ custom basis sets
basis_external /path/to/basisset        # no spaces in path allowed
dkho 2

# job 1 for singlet+triplet; job 2 for doublets
jobs 1 2 1

occ 14 13       # for job 1 and 2
closed 11 10    # for job 1 and 2

nelec           24 23 24   # for job 1
nelec           24 23 24   # for job 2

roots           4 0 3   # for job 1
roots           0 2 0   # for job 2

rootpad         1 0 1   # for job 1
rootpad         0 2 0   # for job 2
```

This template specifies two jobs, where job 1 should be used to compute singlet and triplet states, and job 2 used for doublet states. Job 1 is a SA(4S+3T)-CASSCF(2,3) computation, with singlets and triplets each having 24 electrons. Job 2 is a SA(2D)-CASSCF(3,3) computation, with 23 electrons. Note how for different jobs it is possible to have different active spaces and state-averaging schemes. However, keep in mind that all states of a given multiplicity are always calculated in the same job (e.g., it is not possible to have one job for S_0 and another job for S_1 and S_2).

It is also possible to do a mixed input, for example having two jobs, but only giving one number after **occ** or **closed**. The interface provides comprehensive error messages during the template check.

Also note the **basis_external** keyword. It provides a file, whose content is used in the basis set definition (it is inserted verbatim into **basis={...}** in the MOLPRO input). It is possible to use the generated input from the Basis Set Exchange Libraryhttps://bse.pnl.gov/bse/portalBasis Set Exchange Library, but the **basis={** and **}** need to be deleted from the file.

Remember that **molpro_input.py** cannot create multi-job templates or templates with the **basis_external** keyword.

6.3.2 Resource file: MOLPRO.resources

The interface requires some additional information beyond the content of **QM.in**. This information is given in the file **MOLPRO.resources**, which must reside in the directory where the interface is started. This file uses a simple "**keyword argument**" syntax. Comments using # and blank lines are possible, the order of keywords is arbitrary. Lines with unknown keywords are ignored, since the interface just searches the file for certain keywords.

Table ?? lists the existing keywords. A fully commented resource file for this interface with all possible options is located in **$SHARC/../examples/SHARC_MOLPRO/**.

Mandatory keywords are the paths to MOLPRO, the scratch directory, and to the WFOVERLAP executable (the latter for overlap, Dyson, or NACDR calculations).

Parallel execution of MOLPRO In the new version of the SHARC-MOLPRO interface, calculations of multiple independent active spaces ("jobs"), of several gradients, and of several nonadiabatic coupling vectors are automatically parallelized over the given number of CPU cores.

Note that in the new version of the interface, MOLPRO calls itself cannot be run with multiple CPUs.

6.3.3 Error checking

The interface is written such that the output of MOLPRO is checked for commonly occuring errors, mostly bad convergence in the MCSCF or CP-MCSCF parts. In these cases, the input is adjusted and MOLPRO restarted. This will be done until all calculations are finished or an unrecoverable error is detected. The interface will try to solve the following error messages:

Table 6.5: Keywords for the **`MOLPRO.resources`** input file.

Keyword	Description
`molpro`	Is followed by a string giving the path to the MOLPRO directory. This directory should contain the executable **`molpro.exe`**. Relative and absolute paths, environment variables and ~ can be used.
`scratchdir`	Is a path to the temporary directory. Relative and absolute paths, environment variables and ~ can be used. If the directory does not exist, the interface will create it. In any case, the interface will delete this directory after the calculation.
`savedir`	Is a path to another temporary directory. Relative and absolute paths, environment variables and ~ can be used. The interface will store files needed for restart there.
`memory`	(int) Memory for MOLPRO and WFOVERLAP in MB.
`ncpu`	(int) Number of CPU cores for parallel computation of gradients/NAC vectors.
`delay`	(float) Time in seconds between starting parallel MOLPRO runs. Useful to lessen the I/O burden when having many runs starting at the same time.
`gradaccudefault`	(float) Default accuracy for CP-MCSCF.
`gradaccumax`	(float) Worst acceptable accuracy for CP-MCSCF (see below).
`always_orb_init`	Do not use the orbital guesses from previous calculations/time steps, but always use the provided initial orbitals.
`always_guess`	Always use the orbital guess from MOLPRO's guess module.
`wfoverlap`	Is the path to the WFOVERLAP executable. Relative and absolute paths, environment variables and ~ can be used.
`numfrozcore`	Number of neglected core orbitals for overlap and Dyson calculations.
`numocc`	Number of doubly occupied orbitals for Dyson calculations.
`nooverlap`	Do not save determinant files for overlap computations.
`debug`	Increases the verbosity of the interface.
`no_print`	Reduces interface standard output.

EXCESSIVE GRADIENT IN CI This error message can occur in the MCSCF part. The calculation is restarted with a P-space threshold (see MOLPRO manual) of 1. If the error remains, the threshold is quadrupled until the calculation converges or the threshold is above 100.

NO CONVERGENCE IN REFERENCE CI The error occurs in the CI part. The calculation is restarted with a P-space threshold (see MOLPRO manual) of 1. If the error remains, the threshold is quadrupled until the calculation converges or the threshold is above 100.

NO CONVERGENCE OF CP-MCSCF This error occurs when solving the linear equations needed for the calculation of MCSCF gradients or nonadiabatic coupling vectors. In this case, the interface finds in the output the value of closest convergence and restarts the calculation with the value found as the new convergence criterion. This ensures that the CP-MCSCF calculation converges, albeit with lower accuracy for this gradient for this time step.

This error check is controlled by two keywords in the **`MOLPRO.resources`** file. The interface first tries to converge the CP-MCSCF calculation to **`gradaccudefault`**. If this fails, it tries to converge to the best value possible within 900 iterations. **`gradaccumax`** defines the worst accuracy accepted by the interface. If a CP-MCSCF calculation cannot be converged below **`gradaccumax`** then the interface exits with an error, leading to the abortion of the trajectory.

6.3.4 Things to keep in mind

Initial orbital guess For CASSCF calculations it is always a good idea to start from converged MOs from a nearby geometry. For the first time step, if a file **`QM/wf.init`** is present, the SHARC-MOLPRO interface will take this file for the starting orbitals. In case of multi-job calculations, separate initial orbitals can be provided with files called **`QM/wf.<job>.init`**, where **`<job>`** is an integer. In subsequent calculations, the MOs from the previous step will be used (unless the **`always_orb_init`** keyword is used).

Basis sets Note that the MOLPRO interface cannot calculation SA-CASSCF gradients for generally contracted basis sets (like Dunning's "cc" basis sets or Roos' "ANO" basis sets). Only segmented basis sets are allowed, like the Pople basis sets and the "def" family from TURBOMOLE.

In order to employ user-defined basis sets, in the template the keyword **`basis_external`**, followed by a filename, can be used. The interface will then take the content of this file and insert it as basis set definition in the MOLPRO input files (i.e., it will add in the input **`basis={ <content of the file> }`**. Note that the filename must not contain spaces.

6.3.5 Molpro input generator: `molpro_input.py`

In order to quickly setup simple inputs for MOLPRO, the SHARC suite contains a small script called **`molpro_input.py`**. It can be used to setup single point calculations, optimizations and frequency calculations on the HF, DFT, MP2 and CASSCF level of theory. Of course, MOLPRO has far more capabilities, but these are not covered by **`molpro_input.py`**. However, **`molpro_input.py`** can also prepare template files which are compatible with the SHARC-MOLPRO interface (**`MOLPRO.template`** file).

The script interactively asks the user to specify the calculation and afterwards writes an input file and optionally a run script.

Input

Type of calculation Choose to either perform a single-point calculation or a minimum optimization (including optionally frequency calculation), to generate a template file, or an optimization of a state crossing. For the template generation, no geometry file is needed, but the script looks for a **`MOLPRO.input`** in the same directory and allows to copy the settings.

For single-point calculations, optimizations and frequency calculations, files in MOLDEN format called **`geom.molden`**, **`opt.molden`** or **`freq.molden`**, respectively, are created (containing the orbitals, optimization steps and normal modes, respectively). The file **`freq.molden`** can be used to generate initial conditions with **`wigner.py`**.

Geometry Specify the geometry file in xyz format. Number of atoms and total nuclear charge is detected automatically. After the user inputs the total charge, the number of electrons is calculated automatically.

In the case of the generation of a template file, instead only the number of electrons is required.

Non-default atomic masses If a frequency calculation is requested, the user may modify the mass of specific atoms (e.g. to investigate isotopic effects). In the following menu, the user can add or remove atoms with their mass to a list containing all atoms with non-default masses. Each atom is referred to by its number as in the geometry file. Using the command **`show`** the user can display the list of atoms with non-default masses. Typing **`end`** confirms the list.

Note that when using the produced MOLDEN file later with **`wigner.py`**, the user has to enter the same non-default masses again, since the MOLDEN file does not contain the masses and **`wigner.py`** has no way to retrieve these numbers.

Level of theory Supported are Hartree-Fock (HF), density functional theory (DFT), Møller-Plesset perturbation theory (MP2), equation-of-motion coupled-cluster with singles and doubles (EOM-CCSD) and CASSCF (either single-state or state-averaged). All methods (except EOM-CCSD) are compatible with odd-electron wave functions (**`molpro_input.py`** will use the corresponding UHF, UMP2 and UKS keywords in the input file, if necessary).

For template generation, state-average CASSCF is automatically chosen. All methods (except EOM-CCSD) can be combined with optimizations and frequency calculations, however, the frequency calculation is much more efficient with HF or SS-CASSCF.

DFT functional For DFT calculations, enter a functional and choose whether dispersion correction should be applied. Note that the functional is just a string which is not checked by **`molpro_input.py`**.

Basis set The basis set is just a string which is not checked by **`molpro_input.py`**.

CASSCF settings For CASSCF calculations, enter the number of active electrons and orbitals.

For SS-CASSCF, only the multiplicity needs to be specified. For SA-CASSCF, specify the number of states per multiplicity to be included. Note that MOLPRO allows to average over states with different numbers of electrons. This feature is not supported in **molpro_input.py**. However, the user can generate a closely-matching input and simply add the missing states to the CASSCF block manually.

For optimizations at SA-CASSCF level, the state to be optimized has to be given. For crossing point optimizations, two states need to be entered. The script automatically detects whether a conical intersection or a crossing between states of different multiplicity is requested and sets up the input accordingly.

EOM-CCSD settings EOM-CCSD calculations allow to calculate relatively accurate excited-state energies and oscillator strengths from a Hartree-Fock reference, but only for singlet excited states.

The user has to specify the number of states to be calculated. If only one state is requested, the script will setup a regular ground state CCSD calculation, while for more than one states, an EOM-CCSD calculation is setup. Note that the calculation of transition properties takes twice as long as the energy calculation itself.

Memory Enter the amount of memory for MOLPRO. Note that values smaller than 50 MB are ignored, and 50 MB are used in this case.

Run script If requested, the script also generates a simple Bash script (**run_molpro.sh**) to directly execute MOLPRO. The user has to enter the path to MOLPRO and the path to a suitable (fast) scratch directory.

Note that the scratch directory will be deleted after the calculation, only the wave function file **wf** will be copied back to the main directory.

6.4 MOLCAS Interface

The SHARC-MOLCAS interface can be used to conduct excited-state dynamics based on MOLCAS' CASSCF wave functions, and with CASPT2 or MS-CASPT2 energies (and numerical gradients). RASSCF wave functions are not supported currently. It is important to note that currently, SHARC was only tested to work with OPENMOLCAS 18, which is freely available. MOLCAS versions 8.3 and 8.4 should also work, but no guarantee is given. Note that within this Manual, MOLCAS and OPENMOLCAS are used synonymously, because from SHARC's viewpoint, they only differ in the driver program (**pymolcas** or **molcas.exe**).

Important note: please make sure that a copy of the respective driver program (**pymolcas** or **molcas.exe**) is located in the **$MOLCAS/bin/** directory, as the interface will search this directory for these programs to distinguish between MOLCAS and OPENMOLCAS.

The interface uses the modules GATEWAY, SEWARD (integrals), RASSCF (wave function, energies), RASSI (transition dipole moments, spin-orbit couplings, overlaps), MCLR, and ALASKA (gradients). For CASPT2 and MS-CASPT2, numerical gradients can be calculated, where the interface itself controls the calculations at the displaced geometries. The interface can also numerically differentiate dipole moments and spin-orbit couplings. Since MOLCAS is not able to calculate the full nonadiabatic coupling vectors, only wave function overlaps are currently implemented in the interface (overlaps are calculated via RASSI). Using the WFOVERLAP code, it is possible to calculate Dyson norms between neutral and ionic states. Note that (unlike MOLPRO) MOLCAS cannot average over states of different multiplicities; hence, the multiplicities are always computed in separate jobs which all share the same CAS settings.

The SHARC-MOLCAS interface furthermore allows to perform QM/MM dynamics (CASSCF plus force fields), through the MOLCAS-TINKER interface.

The interface needs two additional input files, a template file for the quantum chemistry (file name is **MOLCAS.template**) and a resource file (**MOLCAS.resources**). If the interface finds files with the name **QM/MOLCAS.<i>.JobIph.init** or **QM/MOLCAS.<i>.RasOrb.init**, they are used as initial wave function files, where **<i>** is the multiplicity. In the case of QM/MM calculations, two more input files are needed: **MOLCAS.qmmm.key**, which contains the path to the force field parameters and the partitioning into QM and MM regions, and **MOLCAS.qmmm.table**, which contains the connectivity and force field IDs per atom.

6.4.1 Template file: `MOLCAS.template`

This file contains the specifications for the wave function. Note that this is not a valid MOLCAS input file. No sections like **`$GATEWAY`**, etc., can be used. The file only contains a number of keywords, given in table **??**. The actual input files are automatically generated.

A fully commented template file with all possible options is located in **`$SHARC/../examples/SHARC_MOLCAS/`**.

Simple template files can be setup with the tool **`molcas_input.py`**.

6.4.2 Resource file: `MOLCAS.resources`

The file **`MOLCAS.resources`** contains mainly paths (to the MOLCAS executables, to the scratch directory, etc.). This file must reside in the same directory where the interface is started. It uses a simple "**`keyword argument`**" syntax. Comments

Table 6.6: Keywords for the **`MOLACS.template`** file.

Keyword	Description
basis	The basis set used. Note that some basis sets (e.g., Pople basis sets) do not work, since the spin-orbit integrals cannot be calculated.
baslib	Can be used to provide the path to a custom basis set library (analogous to the baslib keyword in MOLCAS).
nactel	Number of active electrons for CASSCF.
ras2	Number of active orbitals for CASSCF.
inactive	Number of inactive orbitals.
roots	Followed by a list of integers, giving the number of states per multiplicity in the state-averaging procedure.
rootpad	Followed by a list of integers, giving the number of extra, zero-weight states in the state-averaging.
method	Followed by a string, which is either "CASSCF", "CASPT2" or "MS-CASPT2" (case insensitive), defining the level of theory. Default is "CASSCF".
no-douglas-kroll	Deactivates the use of the scalar-relativistic DK Hamiltonian and uses a non-relativistic Hamiltonian instead. Default is Douglas-Kroll-Hess 2nd order (In MOLCAS standard parametrization).
ipea	Followed by a float giving the IP-EA shift for CASPT2 (see MOLCAS manual for more information). The default is 0.25, as in MOLCAS.
imaginary	Followed by a float giving the imaginary level shift for CASPT2 (see MOLCAS manual for more information). The default is 0.0, as in MOLCAS.
frozen	Number of frozen orbitals for CASPT2. Default is -1, which lets MOLCAS choose the number of frozen orbitals.
cholesky	Activates Cholesky decomposition in MOLCAS. Recommended for large basis sets. Will use numerical gradients even for CASSCF. Default is to use regular 4-electron integrals.
cholesky_analytical	Activates analytical SA-CASSCF gradients with Cholesky decomposition. This is only possible with MOLCAS 8.
gradaccudefault	(float) Default accuracy for CP-MCSCF.
gradaccumax	(float) Worst acceptable accuracy for CP-MCSCF.
displ	Cartesian displacement (in Å) for numerical differentiation (default 0.005 Å).
qmmm	Activates the QM/MM mode. In this case, two more input files need to be present (see below).
pcmset	Activates the PCM mode. Three arguments follow: the solvent (a string, default "water"), the AARE value (a float, default 0.4, optional), the RMIN value (a float, default 1.0, optional). Check MOLCAS manual for details.
pcmstate	Defines the state for which the PCM charges will be optimized. Followed by two numbers: multiplicity (1=singlet, ...) and state (1=lowest state of that multiplicity). Default is the first state according to the state request.

using # and blank lines are possible, the order of keywords is arbitrary. Lines with unknown keywords are ignored, since the interface just searches the file for certain keywords.

Table ?? lists the existing keywords. A fully commented resource file for this interface with all possible options is located in **$SHARC/../examples/SHARC_MOLCAS/**.

Note that the interface sets all environment variables necessary to run MOLCAS (e.g., **$MOLCAS**, **$MOLCASMEM**, **$WorkDir** ,**$Project**) automatically, based on the input from **MOLCAS.resources** and **QM.in**.

Some explanations on parallelization: There are two parallel modes in which the interface can act. In the first mode, the wave function and expectation values are calculated serially first, and then all analytical gradients are computed in parallel, over the given number of CPUs. If the number of CPUs is one, then still the interface prepares independent computations for each gradient, and runs those sequentially on one CPU. If the given number of CPUs is negative or zero, then the gradients are not prepared as separate MOLCAS calculations, but together with the other quantities. The latter variant has less overhead, but is only recommended if the user is sure that the gradient calculations (MCLR) will converge in virtually all cases. If MCLR does not converge occasionally, it is advisable to use NCPU=1 instead of NCPU≤0.

In the second parallelization mode, used for numerical gradients, the central point is calculated first and then all displacements are computed in parallel. This mode will be used for gradients on CASPT2, MS-CASPT2 and Cholesky-CASSCF level as well as for spin-orbit/dipole moment derivatives on all levels. In this mode there is no distinction between NCPU=1 and NCPU≤0.

Table 6.7: Keywords for the **MOLCAS.resources** file.

Keyword	Description
molcas	Is the path to the OPENMOLCAS installation. This directory should contain subdirectories like **bin/**, **basis_library/**, **data/**, or **lib/**. Relative and absolute paths, environment variables and ~ can be used. The interface will set **$MOLCAS** to this path. If this keyword is not present in **MOLCAS.resources**, the interface will use the environment variable **$MOLCAS**, if it is set.
tinker	The path to the TINKER installation directory, usable with MOLCAS. This directory should contain the **bin/** subdirectory.
wfoverlap	Is the path to the WFOVERLAP executable. Relative and absolute paths, environment variables and ~ can be used. Only needed for Dyson norm calculations.
scratchdir	Is a path to the temporary directory. Relative and absolute paths, environment variables and ~ can be used. If it does not exist, the interface will create it. In any case, the interface will delete this directory after the calculation.
savedir	Is a path to another temporary directory. Relative and absolute paths, environment variables and ~ can be used. The interface will store files needed for restart there.
memory	The memory usable by MOLCAS. The interface will set **$MOLCASMEM** to this value. The default is 10 MB.
ncpu	The number of CPUs used by the interface. If zero or negative, one CPU will be used and all subtasks (Hamiltonian, dipole moments, analytical gradients, overlaps) will be done in one MOLCAS call. If positive, analytical gradients will be calculated by separate calls to MOLCAS, parallelized over the given number of CPUs.
mpi_parallel	Uses MPI parallel MOLCAS runs. The number of cores is dynamically chosen based on the available cores and the number of tasks (energies, gradients, displacements).
schedule_scaling	Gives the expected parallelizable fraction of the MOLCAS run time (Amdahl's law). With a value close to zero, the interface will try to run all jobs at the same time. With values close to one, jobs will be run sequentially with the maximum number of cores.
delay	Followed by a float giving the delay in seconds between starting parallel jobs to avoid excessive disk I/O.
always_orb_init	Do not use the orbital guesses from previous calculations/time steps, but always use the provided initial orbitals.
always_guess	Always use the orbital guess from MOLCAS's **guessorb** module.
debug	Increases the verbosity of the interface.
no_print	Reduces interface standard output.

6.4.3 Template file generator: `molcas_input.py`

This is a small interactive script to generate template files for the SHARC-MOLCAS interface. It simply queries the user for some input parameters and then writes the file **`MOLCAS.template`**, which can be used to run SHARC simulations with the SHARC-MOLCAS interface. The input generator can also be used to write proper MOLCAS input files for single-point calculations and optimizations/frequency calculations on CASSCF and (MS-)CASPT2 level of theory.

Type of calculation Choose to either perform a single-point calculation or a minimum optimization (including optionally frequency calculation), or to generate a template file. For the template generation, no geometry file is needed, but the script looks for a **`MOLCAS.input`** in the same directory and allows to copy the settings.

For single-point calculations, optimizations and frequency calculations, files in MOLDEN format are created (containing the orbitals, optimization steps and normal modes, respectively). The file **`MOLCAS.freq.molden`** can be used to generate initial conditions with **`wigner.py`**.

Geometry file The geometry file is only used to calculate the nuclear charge.

Charge This is the overall charge of the molecule. This number is used with the nuclear charge to calculate the number of electrons.

Method Choose either CASSCF or CASPT2. Multi-state CASPT2 can be requested later.

Basis set This is simply a string, which is *not* checked by the script to be a valid basis set of the MOLCAS library.

Number of active electrons and orbitals These settings are necessary for the definition of the CASSCF wave function. The number of inactive orbitals is automatically calculated from the total number of electrons and the number of active electrons.

States for state-averaging For each multiplicity, the number of states for the state-averaging procedure must be equal or larger than the number of states used in the dynamics.

Further settings Depending on the run type and method, the script might ask further questions regarding the root to optimize, CASPT2 settings (whether to do multi-state CASPT2, IPEA shift, imaginary level shift), or whether a spin-orbit RASSI should be performed (for input file generation only).

6.4.4 QM/MM key file: `MOLCAS.qmmm.key`

This file defines some aspects of the QM/MM calculation. An example is given below:

```
parameters $MOLCAS/tinker/params/amber99sb.prm

QMMM 18
QM    -1 15
MM   -16 18

QMMM-ELECTROSTATICS ESPF
```

The first line defines the force field parameters, which are read from TINKER's library here. Note that in the file path, environment variables are expanded by the interface. The next three lines define the QM and MM regions of the system, where in this case atoms 1–15 are in the QM region and atoms 16–18 are in the MM region (note the minus signs indicating the begin of an interval). The last line defines how to treat electrostatics in the calculation (currently, only ESPF is implemented for MOLCAS, so this line has to be present as shown, or omitted).

6.4.5 QM/MM connection table file: `MOLCAS.qmmm.table`

This is the template to generate the TINKER xyz file, which besides the geometry contains the force field atom type number and the connectivity information. A sample looks like:

```
8
1 N*   1.6  3.3 -2.0  1132  2
2 CK   0.3  3.1 -2.6  1136  1  3  4
3 H5  -0.4  2.9 -1.9  1145  2
4 NB   0.3  3.2 -3.9  1135  2  5
5 CB   1.6  3.5 -4.2  1134  4  6
6 CA   2.2  3.8 -5.4  1140  5  7
7 N2   1.6  3.8 -6.6  1142  6  8
8 H    2.1  3.9 -7.5  1143  7
```

The first line contains the number of atoms. In each following line, the atom index, the atomic symbol (or name), the coordinates of the atom, the force field atom type number and the index of atoms bonded to the current atom. The coordinates in this file are not used and are replaced by the SHARC-MOLCAS interface by the actual coordinates of the atoms. The atom type number can be looked up in the **`.prm`** files of the TINKER parameter directory.

6.5 COLUMBUS Interface

The SHARC-COLUMBUS interface allows to run SHARC dynamics based on COLUMBUS' CASSCF, RASSCF and MRCI wave functions. The interface is compatible to COLUMBUS calculations utilizing the COLUMBUS-MOLCAS interface (SEWARD integrals and ALASKA gradients), or using the DALTON integral code distributed with COLUMBUS. Using SEWARD integrals, spin-orbit couplings can be calculated, but no nonadiabatic couplings (only overlaps can thus be used). Using DALTON integrals, spin-orbit couplings are not possible, but nonadiabatic couplings can be calculated. The CASSCF step can be done with either COLUMBUS' **`mcscf`** code (all features available) or with MOLCAS' **`rasscf`** code (faster, but no gradients possible). The interface utilizes the WFOVERLAP program to calculate the overlap matrices. The interface can also calculate Dyson norms between neutral and ionic wave functions using the WFOVERLAP code.

The interface needs as additional input the file **`QM/COLUMBUS.resources`** and a template directory containing all input files needed for the COLUMBUS calculations. Initial MOs can be given in the file **`QM/mocoef_mc.init`**. For multiple jobs, initial MOs can be given as **`QM/mocoef_mc.init.<job>`**. For runs with **`rasscf`**, initial MOs have to be given as **`molcas.RasOrb.init`** or **`molcas.RasOrb.init.<job>`**.

6.5.1 Template input

The interface does not generate the full COLUMBUS input on-the-fly. Instead, the interface uses an existing set of input files and performs only necessary modifications (e.g., the number of states). The set of input files must be provided by the user. Please see the COLUMBUS online documentationhttp://www.univie.ac.at/columbus/docs$_{C}OL70/documentation_{m}ain.html$COLUMBUSor
and, most importantly, the COLUMBUS SOCI tutorialhttp://www.univie.ac.at/columbus/docs$_{C}OL70/tutorial-SO.pdf$COLUMBUS$SOCIt$
for a documentation of the necessary input. The SHARC tutorial also has a section about generating the required COLUMBUS input file collection. An example template directory is located in **`$SHARC/../examples/SHARC_COLUMBUS/`**.

Generally, the input consists of a directory with one subdirectory with input for each multiplicity (singlets, doublets, triplets, ...). However, even-electron wave functions of different multiplicities can be computed together in the same job if spin-orbit couplings are desired. Independent multiple-DRT inputs (ISC keyword) are also acceptable. Note that symmetry is not allowed when using the interface.

The path to the template directory must be given in **`COLUMBUS.resources`**, along with the other resources settings.

Integral input The interface is able to use input for calculations using SEWARD or DALTON integrals. If you want to calculate SOCs, you have to use SEWARD and have to include the AMFI keyword in the integral input. If you use DALTON, SOCs are not available, but it is possible to compute nonadiabatic couplings.

It is important to make sure that **the order of atoms** in the template input files and in the SHARC input **is consistent**.

Furthermore, it is necessary to prepare all template subdirectories with the same integral code and the same AO basis set.

MCSCF input The MCSCF section can use any desired state-averaging scheme, since the number of states in MCSCF is independent of the number of states in the MRCI module. However, frozen core orbitals in the MCSCF step are not possible (since otherwise gradients cannot be computed). Prepare the MCSCF input for CI gradients. It is advisable to use very tight MCSCF convergence criteria.

If gradients are not needed, you can also manually prepare a MOLCAS RASSCF input in **molcas.input**, in order to use MOLCAS RASSCF instead of COLUMBUS MCSCF (see **molcas_rasscf** keyword).

MRCI input Either prepare a single-DRT input without SOCI (to cover a single multiplicity), a single-DRT input with SOCI and a sufficient maximum multiplicity for spin-orbit couplings or an independent multiple-DRT input (as, e.g., for ISC optimizations). Make sure that all multiplicities are covered with all input directories.

In the MRCI input, make sure to use sequential **ciudg**. Also take care to setup gradient input on MRCI level.

Job control Setup a single-point calculation with the following steps:

- SCF
- MCSCF
- MR-CISD (serial operation) **or** SO-CI coupled to non-rel CI (for SOCI DRT inputs)
- one-electron properties for all methods
- transition moments for MR-CISD
- nonadiabatic couplings (and/or gradients)

Request first transition moments and interstate couplings (or alternatively full nonadiabatic couplings if Dalton integrals are used) in the following dialogues. Analysis in internal coordinates and intersection slope analysis are not required.

6.5.2 Resource file: COLUMBUS.resources

Beyond the information from **QM.in** the interface needs additional input, which is read from the file **COLUMBUS.resources**. Table ?? lists the keywords which can be given in the file. A fully commented resource file with all possible options is located in **$SHARC/../examples/SHARC_COLUMBUS/**.

Table 6.8: Keywords for the **COLUMBUS.resources** file.

Keyword	Description
`columbus`	Path to the COLUMBUS main directory.This directory should contain executables like **runc**, **mcscf.x**, **cidrt.x**, or **ciudg.x**. Relative and absolute paths, environment variables and ~ can be used.
`molcas`	Path to the MOLCAS main directory. Relative and absolute paths, environment variables and ~ can be used. This path is only used for overlap/Dyson calculations (since in this case the interface calls MOLCAS explicitly). Otherwise COLUMBUS will use the path to MOLCAS specified during the installation of COLUMBUS.
`wfoverlap`	Path to the wave function overlap and Dyson norm code. Relative and absolute paths, environment variables and ~ can be used. Only necessary if overlaps or Dyson norms are calculated. Needs a suitable code that computes overlaps of CI wave functions.
`runc`	Path to the **runc** script for COLUMBUS execution. Default is **$COLUMBUS/runc**. This keyword is intended for users who like to modify **runc**.
`scratchdir`	Path to the temporary directory, used to perform the COLUMBUS calculations. Relative and absolute paths, environment variables and ~ can be used. If it does not exist, the interface will create it. The interface will delete this directory after the calculation.
`savedir`	Path to another directory. Relative and absolute paths, environment variables and ~ can be used. The interface will store files needed for restart in this directory. Default is a subdirectory of the directory where the interface is executed.

Continued on next page

Table 6.8 – Continued from previous page

Keyword	Description
`memory`	(integer, MB) Memory for COLUMBUS. The maximum amount of memory used by the wfoverlap code is also controlled with this keyword.
`ncpu`	(integer) Number of CPU cores to use for SMP-parallel execution of the wfoverlap code. Parallel COLUMBUS is currently not supported.
`wfthres`	(float) Gives the amount of wave function norm which will be kept in the truncation of the determinant files for Dyson and overlap calculations. Default is 0.97 (i.e., the wave functions in the determinant files will have a norm of 0.97)
`nooverlap`	(no argument) Do not keep the files necessary to calculate wave function overlaps in the next time step. If Dyson norms are requested, **`nooverlap`** is ignored.
`always_orb_init`	Use the initial MO guess (**`mocoef_mc.init`**) for all time steps, not only for the first step.
`always_guess`	In all time steps obtain the MO guess from an SCF calculation, instead of using the MOs from the previous step.
`integrals`	Followed by a string which is either **`seward`** or **`dalton`**. Chooses the integral program for COLUMBUS. Note that with DALTON integrals SOC is not available. Default is **`seward`**.
`molcas_rasscf`	Use MOLCAS' RASSCF program instead of COLUMBUS' MCSCF program. Needs a properly prepared COLUMBUS input (**`&RASSCF`** section in **`molcas.input`**). Note that gradients are not available in this mode.
`template`	Is followed by the path to the directory containing the template subdirectories. Relative and absolute paths, environment variables and ~ can be used. See also ??.
`DIR`	See ??.
`MOCOEF`	See ??.
`numfrozcore`	Can be used to override the number of frozen orbitals in Dyson calculations (Default is the value from the **`cidrt`** input).
`numocc`	Can be used to declare orbitals above the frozen orbitals as doubly occupied in Dyson calculations. No ionization from these orbitals is calculated, but otherwise they are considered in the Dyson calculations. Default is zero.
`debug`	Increases the verbosity of the interface.
`no_print`	Reduces interface standard output.

6.5.3 Template setup

The template directory contains several subdirectories with input for different multiplicities. An example is given in figure ??. In **`COLUMBUS.resources`**, the user has to associate each multiplicity to a subdirectory. The line "**`DIR 1 Sing_Trip`**" would make the interface use the input files from the subdirectory **`Sing_Trip`** when calculating singlet states (the **`1`** refers to singlet calculations). All calculations using a particular input subdirectory are called a job.

Additionally, the user must specify which job(s) provide the MO coefficients (e.g., the calculation for doublet states could be based on the same MOs as the singlet and triplet calculation). The line "**`MOCOEF Doub_Quar Sing_Trip`**" would tell the interface to do a MCSCF calculation in the **`Sing_Trip`** job, and reuse the MOs when doing the **`Doub_Quar`** job without reoptimizing the MOs.

6.6 Analytical PESs Interface

The SHARC suite also contains an interface which allows to run dynamics simulations on PESs expressed with analytical functions. In order to allow dynamics simulations in the same way as it is done on-the-fly, the interface uses two kinds of potential couplings:

- couplings that are pre-diagonalized in the interface (yielding the equivalent of the MCH basis),
- couplings given by the interface to SHARC as off-diagonal elements.

The interface needs one additional input file, called **`Analytical.template`**, which contains the definitions of all analytical expressions.

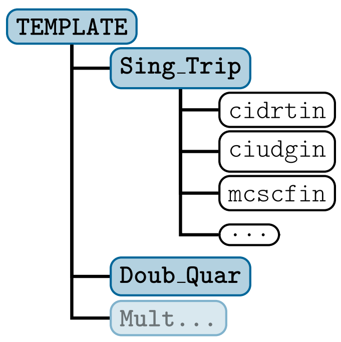

Figure 6.2: Example directory structure of the COLUMBUS template directory

6.6.1 Parametrization

The interface has to be provided with analytical expressions for all matrix elements of the following matrices in the diabatic basis:

- Hamiltonian: $\mathbf{H}$
- Derivative of the Hamiltonian with respect to each atomic coordinate: $\mathbf{H}_{x_i}$
- (Transition) dipole matrices for each polarization direction: $\mathbf{M}_i$
- Real and imaginary part of the SOC matrix: Σ
- (Optionally) the derivatives of the transition dipole matrices: $\mathbf{D}_{i,x_j}$

The diabatic Hamiltonian is diagonalized:

$$\mathbf{H}^{\mathrm{d}} = \mathbf{W}^{\dagger}\mathbf{H}\mathbf{W} \tag{6.1}$$

Then the following calculations lead to the MCH matrices which are passed to SHARC:

$$\mathbf{H}^{\mathrm{MCH}} = \mathbf{H}^{\mathrm{d}} + \mathbf{W}^{\dagger}\Sigma\mathbf{W} \tag{6.2}$$

$$\left(\mathbf{g}_{\alpha}^{\mathrm{MCH}}\right)_{x_i} = \left(\mathbf{W}^{\dagger}\mathbf{H}_{x_i}\mathbf{W}\right)_{\alpha\alpha} \tag{6.3}$$

$$\mathbf{M}_i^{\mathrm{MCH}} = \mathbf{W}^{\dagger}\mathbf{M}_i\mathbf{W} \tag{6.4}$$

$$\mathbf{S}^{\mathrm{MCH}}(t_0, t) = \mathbf{W}^{\dagger}(t_0)\mathbf{W}(t) \tag{6.5}$$

$$\mathbf{D}_{i,x_j}^{\mathrm{MCH}} = \mathbf{W}^{\dagger}\mathbf{D}_{i,x_j}\mathbf{W} \tag{6.6}$$

The MCH Hamiltonian is the diagonalized diabatic Hamiltonian plus the SO matrix transformed into the MCH basis. The gradients in the MCH basis are obtained by transforming the derivative matrices into the MCH basis. The dipole matrices are also simply transformed into the MCH basis. The overlap matrix is the overlap of old and new transformation matrix.

6.6.2 Template file: `Analytical.template`

The interface-specific input file is called **`Analytical.template`**. It contains the analytical expressions for all matrix elements mentioned above. All analytical expressions in this file are evaluated considering the atomic coordinates read from **`QM.in`**.

The file consists of a file header and the file body. The file body consists of variable definition blocks and matrix blocks. A commented template file is located in **`$SHARC/../examples/SHARC_Analytical/`**.

Header The header looks similar to an xyz file:

```
2
2
I       xI      0       0
Br      xBr     0       0
```

Here, the first line gives the number of atoms and the second line the number of states.

On the remaining lines, each Cartesian component of the atomic coordinates is associated to a variable name, which can be used in the analytical expressions. If a zero (**0**) is given instead of a variable name, then the corresponding Cartesian coordinate is neglected. In the above example, the variable name **xI** is associated with the x coordinate of the first atom given in **QM.in**. The y and z coordinates of the first atom are neglected.

All variable names must be valid Python identifiershttps://docs.python.org/2/reference/lexical$_a$*nalysis.html#identifiersvalidPythoni*
and must not start with an underscore. Hence, all strings starting with a letter, followed by an arbitrary number of letters, digits and underscores, are valid. It is not allowed to use a variable name twice.

Note that the file header also contains the atom labels, which are just used for cross-checking against the atom labels in **QM.in**.

The file header must not contain comments, neither at the end of a line nor separate lines. Also, blank lines are not allowed in the header. After the last line of the header (where the variables for the n_{atom}-th atom are defined), blank lines and comments can be used freely (except in matrix blocks).

Variable definition blocks Variable definition blocks can be used to store additional numerical values (beyond the atomic coordinates) in variables, which can then be used in the equations in the matrix blocks. The most obvious use for this is of course to define values which will appear several times in the equations.

A variable definition block looks like:

```
Variables
A1      0.067
g1      0.996   # Trailing comment
# Blank line with comment only
R1      4.666
End
```

Each block starts with the keyword "Variables" and is terminated with "End". Inbetween, on each line a variable name and the corresponding numerical value (separated by blanks) can be given. Note that the naming conventions given above also apply to variables defined in these blocks.

There can be any number of complete variable definitions blocks in the input file. All blocks are read first, before any matrix expressions are evaluated. Hence, the relative order of the variable blocks and the matrix blocks does not matter. Also, note that variable names must not appear twice, so variables cannot be redefined halfway through the file.

Matrix blocks The most important information in the input file are of course contained in the expressions in the matrix blocks. In general, a matrix block has the following format:

```
Matrix_Identifier
V11
V12,    V22
V13,    V23,    V33
...
```

The first line identifies the type of matrix. Those are valid identifiers:

Hamiltonian	Defines the Hamiltonian including the diabatic potential couplings.
Derivatives followed by a variable name	Derivative of the Hamiltonian with respect to the given variable.
Dipole followed by 1, 2 or 3	(Transition) dipole moment matrix for Cartesian direction x, y or z, respectively.
Dipolederivatives followed by 1, 2 or 3 followed by a variable name	Derivative of the respective dipole moment matrix.
SpinOrbit followed by **R** or **I**	Real or Imaginary (respectively) part of the spin-orbit coupling matrix Σ.

Since the interface searches the file for these identifiers starting from the top until it is found, for each matrix only the first block takes effect. Note that the Hamiltonian and all relevant derivatives must be present. If dipole matrix, dipole derivative matrix or SO matrix definitions are missing, they will be assumed zero.

In the lines after the identifier, the expressions for each matrix element are given. Note the lower triangular format (all matrices are assumed symmetric, except the imaginary part of the SO matrix, which is assumed antisymmetric). Matrix elements must be separated by commas (so that whitespace can be used inside the expressions). There must be at least as many lines as the number of states (additional lines are neglected). If any line or matrix element is missing, the interface will abort.

An examplary block looks like:

```
Hamiltonian
A1*( (1.-math.exp(g1*(R1-xI+xBr)))**2-1.),
0.0006,                                      3e-5*(xI-xBr)**2
```

It is important to understand that the expressions are directly evaluated by the Python interpreter, hence all expressions must be valid Python expressions which evaluate to numeric (integer or float) values. Only the variables defined above can be used.

Note that exponentiation in Python is `**`. In order to provide most usual mathematical functions, the **math** module-https://docs.python.org/2/library/math.html**math** module is available. Among others, the **math** module provides the following functions:

- **math.exp(x)**: Exponential function
- **math.log(x)**: Natural logarithm
- **math.pow(x,y)**: x^y
- **math.sqrt(x)**: $\sqrt{x}$
- **math.cos(x)**, **math.sin(x)**, **math.tan(x)**
- **math.acos(x)**, **math.asin(x)**, **math.atan(x)**
- **math.atan2(y,x)**: $\tan^{-1}\left(\frac{y}{x}\right)$, as in many programming languages (takes care of phases)
- **math.pi**, **math.e**: π and Euler's number
- **math.cosh(x)**, **math.sinh(x)**: Hyperbolic functions (also tanh, acosh, asinh, atanh are available)

6.7 ADF Interface

The SHARC-ADF interface allows to run SHARC simulations with ADF's TD-DFT functionality. The interface is compatible with restricted and unrestricted ground states, but not with symmetry. Spin-orbit couplings are obtained with the perturbative ZORA formalism, and wave function overlaps from the WFOVERLAP code are available (but no nonadiabatic couplings). Dyson norms can also be computed through the WFOVERLAP code. THEODORE (version 2.0 and higher) can be used to perform automatic wave function analysis. QM/MM is possible through ADF's internal implementation, but only mechanical embedding is currently available in ADF.

The interface needs two additional input files, a template file for the quantum chemistry (file name is **ADF.template**) and a resource file (**ADF.resources**). If files **QM/ADF.t21.<job>.init** are are present, they are used to provide an initial orbital guess for the SCF calculation of the respective job.

Before running the ADF interface, it is necessary to source one of the **adfrc** files, so that Python can find the **KFFile** module of ADF.

Note that since SHARC2.1, only ADF2019 and newer are supported by **SHARC_ADF.py**, because older ADF versions do not ship with the **KFFile** module. Also note that **KFFile** requires Python 3 (version 3.5 or higher), hence **SHARC_ADF.py** is a Python 3 script, unlike every other script in the SHARC package.

6.7.1 Template file: ADF.template

This file contains the specifications for the quantum chemistry calculation. Note that this is not a valid ADF input file. The file only contains a number of keywords, given in table ??. The actual input for ADF will be generated automatically through the interface. In order to enable many functionalities of ADF and to allow fine-tuning of the performance for large calculations, the template has a relatively large number of keywords.

A fully commented template file—with all possible options, a comprehensive descriptions, and some practical hints—is located in **`$SHARC/../examples/SHARC_ADF/ADF.template`**. We recommend that users start from this template file and modify it appropriately for their calculations.

6.7.2 Resource file: `ADF.resources`

The file **`ADF.resources`** contains mainly paths (to the ADF executables, to the scratch directory, etc.) and other resources, plus settings for **`wfoverlap.x`** and THEODORE. This file must reside in the same directory where the interface is started. It uses a simple "**`keyword argument`**" syntax. Comments using # and blank lines are possible, the order of keywords is arbitrary. Lines with unknown keywords are ignored, since the interface just searches the file for certain keywords.

Table **??** lists the existing keywords. A fully commented resource file with all possible options and comprehensive descriptions is located in **`$SHARC/../examples/SHARC_ADF/`**.

Parallelization ADF usually shows very good parallel scaling for most calculations. However, it is more efficient to carry out multiple ADF calculations (different multiplicities, multiple gradients) in parallel, each one using a smaller number of CPUs.

In the SHARC-ADF interface, parallelization is controlled by the keywords **`ncpu`** and **`schedule_scaling`**. The first keyword controls the maximum number of CPUs which the interface is allowed to use for all ADF runs simultaneously. The second keyword is the parallel fraction from Amdahl's Law, see Section **??**. With a value close to zero, the interface will try to run all jobs at the same time. With values close to one, jobs will be run sequentially with the maximum number of cores. Typical values for **`schedule_scaling`** are 0.95 for GGA functionals, 0.75 for hybrid functionals, and 0.90 for hybrid functions in combination with the **`rihartreefock`** option.

Note that when using ADF2019 or newer, multiple gradients can be computed in a single ADF run, so that there will be multiple jobs to schedule only if including several multiplicities (beyond singlet plus triplet). If only a single multiplicity is computed (or singlets plus triplets), then the **`schedule_scaling`** keyword does not have any effect.

6.7.3 QM/MM force field file: `ADF.qmmm.ff`

The force field file for ADF contains all force field parameters for the MM part of the QM/MM calculation. The interface uses this file without any modification, except that it will copy the file into the relevant scratch directory.

The force field file needs to be in the format specified in the ADF manualhttps://www.scm.com/doc/QMMM/ADF$_QMMM/The_Force_Fie$ which is similar in format to an AMBER parameter file. **`$SHARC/../examples/SHARC_ADF_qmmm/ADF.qmmm.ff`** contains a functioning example of the file format. Here, we only show briefly the general format:

```
FORCE_FIELD_SETTINGS
=================================
 ELSTAT_1-4_SCALE       0.8333
 ELSTAT_NB_CUTOFF       none
 VDW_1-4_SCALE          0.5
 VDW_DEFAULT_POTENTIAL  1         (1:6-12   2:exp-6  3:exp purely repulsive)
 VDW_NB_CUTOFF          none
 DIELECTRIC_CONSTANT    1.000
=================================



MASSES & ATOM LABELS
-------------------------------
force_field  atomic
 atom_type   symbol    mass        NOTES
================================
    BR         Br     79.9000      bromine
```

Table 6.9: Keywords for the **ADF.template** file.

Keyword	Description
relativistic	If not given, perform a nonrelativistic calculation. Otherwise, copy the line verbatim to the ADF input.
basis	Gives the basis set for all atoms (default SZ).
basis_path	gives the path to the basis library of ADF (~ and $ can be used).
basis_per_element	Followed by an elemental symbol (e.g., "Fe", "H.1") and then by a path to the desired ADF basis set file. Files with frozen core should not be used.
define_fragment	Followed by an elemental symbol (e.g., "Fe", "H.1") and then by a list of atom numbers which should belong to this atom type.
functional	Followed by two strings. First argument gives the type of functional (LDA, GGA, hybrid), second argument gives the functional (VWN, BP86, B3LYP, ...).
functional_xcfun	Enables functional evaluation with the XCFun library within ADF.
dispersion	If present, is written verbatim to ADF input with all arguments.
charge	Sets the total charge of the (QM) system. Can be either followed by a single integer (then the interface will automatically assign the charges to the multiplicities) or by one charge per multiplicity. Automatic assignment might not work correctly for QM/MM computations.
totalenergy	Activates the computation of total energies (by default, ADF computes binding energies). Does not work for relativistic or QM/MM calculations.
cosmo	Followed by a string giving a solvent. Activates COSMO (no gradients possible).
cosmo_neql	Activates non-equilibrium solvation, which is needed for vertical excitation calculations. Is followed by a float giving the square of the refractive index of the solvent.
grid	Followed by two strings (e.g., **beckegrid normal** or **integration 4.0**) defining which integration grid and accuracy to use. For details, see the example template file.
grid_qpnear	Followed by a float giving the maximum distance (in Bohr) an MM point charge can have from a QM atom, such that integration grid points are generated around the MM charge.
grid_per_atom	Followed by a string (e.g., **basic**, **normal**, **good**) and a list of the atoms which should have the given integration accuracy. Can be used multiple times with different qualities.
fit	Followed by two strings (e.g., **zlmfit normal** or **stofit**) defining which Coulomb integration method and accuracy to use. For details, see the example template file.
fit_per_atom	Works like **grid_per_atom**, but for the Coulomb method accuracy.
exactdensity	Enables the **exactdensity** keyword in the ADF input.
rihartreefock	Followed by a quality keyword (e.g., **basic**, **normal**, **good**). If not present, the old HF exchange routines in ADF are used.
rihf_per_atom	Works like **grid_per_atom**, but for the RI Hartree Fock method.
occupations	If present, the foll line is copied verbatim to the ADF input.
scf_iterations	Followed by the maximum number of SCF iterations (default: 100)
linearscaling	Followed by an integer (between 0 and 99), which controls whether certain terms are neglected in ADF.
cpks_eps	Followed by a float (default 0.0001) giving the convergence threshold in the CPKS equations for excited-state gradients.
no_tda	This keyword deactivates TDA, which the interface requests by default.
fullkernel	Uses the full (non-ALDA) kernel in TDDFT. Not compatible with gradients, and automatically activates **functional_xcfun**.
paddingstates	Followed by a list of integers, which give the number of extra states to compute by ADF, but which are neglected in the output. Should not be changed between time steps, as this will break ADF's restart routines.
dvd_vectors	Number of Davidson vectors. Default: min(40,nstates+40).
dvd_tolerance	Energy convergence criterion for the excited states (in Hartree).
dvd_residu	Residual norm convergence criterion for the excited states.
dvd_mblocksmall	Activates the **mblocksmall** keyword in the ADF input. This undocumented keyword changes how the Davidson algorithm works. In reduces computation time, but might lead to incomplete convergence in certain cases.
unrestricted_triplets	Requests that the triplets are calculated in a separate job from an unrestricted ground state. Default is to compute triplets as linear response of the restricted singlet ground state.
modifyexcitations	Followed by an integer, indicating that excitation should only be possible from the first *n* MOs. Can be used to compute core-excitation states (for X-Ray spectra).
qmmm	Activates QM/MM. In this case, the interface will look for two additional input files (see below).
qmmm_coupling	Followed by an integer (1=mechanical embedding, 2=electrostatic embedding). Default is 2.

Table 6.10: Keywords for the **`ADF.resources`** file.

Keyword	Description
`adfhome`	Is the path to the ADF installation. Relative and absolute paths, environment variables and ~ can be used. The interface will set **`$ADFHOME`** to this path, and will also set **`$ADFBIN`** to **`$ADFHOME/bin/`**.
`scmlicense`	Is the path to the ADF license file. Relative and absolute paths, environment variables and ~ can be used.
`scm_tmpdir`	Path to the ADF-internal scratch directory. Usually, this is set at installation, but can be overridden here. Should not exist and must not be identical to scratchdir.
`scratchdir`	Is a path to the temporary directory. Relative and absolute paths, environment variables and ~ can be used. If it does not exist, the interface will create it. In any case, the interface will delete this directory after the calculation.
`savedir`	Is a path to another temporary directory. Relative and absolute paths, environment variables and ~ can be used. The interface will store files needed for restart there.
`memory`	The memory usable by WFOVERLAP. The default is 100 MB. The ADF memory usage cannot be controlled.
`ncpu`	The number of CPUs used by the interface. Is overridden by environment variables from queuing engines (e.g., **`$NSLOTS`** or **`$SLURM_NTASKS_PER_NODE`**). Will either be used to run ADF in parallel or to run several independent ADF runs at the same time.
`schedule_scaling`	Gives the expected parallelizable fraction of the ADF run time (Amdahl's law). With a value close to zero, the interface will try to run all jobs at the same time. With values close to one, jobs will be run sequentially with the maximum number of cores.
`delay`	Followed by a float giving the delay in seconds between starting parallel jobs to avoid excessive disk I/O (usually not necessary).
`qmmm_table`	Followed by the path to the connection table file, in ADF format.
`qmmm_ff_file`	Followed by the path to the force field file, in Amber95-like format for ADF.
`neglected_gradient`	Decides how not-requested gradients are reported back (Options: **`zero`**: default, not-requested gradients are zero; **`gs`**: not-requested gradients are equal to ground state gradient; **`closest`**: not-requested gradients are equal to closest-energy requested gradient).
`wfoverlap`	Path to the WFOVERLAP code. Needed for overlap and Dyson norm calculations.
`wfthres`	(float) Gives the amount of wave function norm which will be kept in the truncation of the determinant files. Default is 0.99 (i.e., the wave functions in the determinant files will have a norm of 0.99). Note that if hybrid functionals and no TDA are used, the response vector can have a norm larger than one, and wfthres should be increased.
`numfrozcore`	Number of frozen core orbitals for overlap and Dyson norm calculations. A value of -1 enables automatic frozen core.
`numocc`	Number of ignored occupied orbitals in Dyson calculations.
`nooverlap`	Do not save determinant files for overlap computations.
`theodir`	Path to the THEODORE installation. Relative and absolute paths, environment variables and ~ can be used. The interface will set **`$PYTHONPATH`** automatically.
`theodore_prop`	Followed by a list with the descriptors which THEODORE should compute. Note that descriptors will only be computed for restricted singlets (and triplets). Instead of a simple list, a Python literal can also be used, as in the THEODORE input files.
`theodore_fragment`	Followed by a list of atom numbers which should constitute a fragment in THEODORE. For multiple fragments, the keyword can be used multiple times. Instead, the keyword can be followed by a Python literal, as in the THEODORE input files.
`always_orb_init`	Do not use the orbital guesses from previous calculations/time steps, but always use the provided initial orbitals.
`always_guess`	Always use the orbital guess from ADF.
`debug`	Increases the verbosity of the interface (standard out). Does not clean up the scratch directory. Copies all ADF outputs to the save directory.
`no_print`	Reduces interface standard output.

```
================================



BONDS   Ebond = 0.5*K(r-ro)**2
--------------------------------
 Atoms    pot
i  -   j  type     K      R       NOTES
==============================
C    CT    1     634.00  1.522    JCC,7,(1986),230; AA
==============================



BENDS    Ebend = 0.5*k(a-ao)^2   (K in kcal/mol*rad^2)
--------------------------------
    Atoms     pot
i  -  j - k  type     K     theta   NOTES
===================================
H1   CT   HC   1    70.00   109.50  M. Swart added
===================================



TORSIONS     Etors=K(1+cos(nt-to))
--------------------------------
    Atoms           pot            per.   shift
i   - j  - k -  l  type     k       n      to      NOTES
=================================================
*    C    CT   *     1     0.0000    2       0.0   JCC,7,(1986),230
=================================================



OUT-OF-PLANE       Etors=K(1-cos(2t-to))
--------------------------------
      Atoms        pot
 i - j  - k  - l   type   K     to     NOTES
=====================================
*    *    CC   *     1   1.00  180.0   HEME improper
=====================================

VAN DER WAALS
--------------------------------
  atom(s)     Emin    Rmin   gamma     NOTES
====================================
    H       0.0157   1.2000  12.00     Ferguson base pair geom.
====================================

CHARGES
--------------------------------
 type      charge(e)    NOTES
=====================
  OW       -0.8340      Amber95 water charges (TIP3P had -0.82)
=====================
```

6.7.4 QM/MM connection table file: **ADF.qmmm.table**

The connection table file defines the topology of the QM/MM system, by defining which atoms belong to QM or MM, and which atoms have MM bonds between them. The file can also be used to define link atoms or to override the MM charges of specific atoms.

In the ADF input file, the interface will automatically add the line **MM_CONNECTION_TABLE** (so this line should not be present in **ADF.qmmm.table**), then copy the content of **ADF.qmmm.table** verbatim to the ADF input, and then add a line **END**. Hence, **ADF.qmmm.table** needs to follow the format required by ADF. See the ADF manualhttps://www.scm.com/doc/QMMM/ADF for details on this input section.

An example of the file is given in **$SHARC/../examples/SHARC_ADF_qmmm/ADF.qmmm.table**. In the simplest form, the content of the file could look like:

```
1       SO      QM      2
2       C       QM      1       3       4
3       H1      QM      2
4       H1      QM      2
5       OW      MM      6      7
6       HW      MM      5
7       HW      MM      5
```

Here, the first column is the atom index, the second is the MM atom type (as defined in the force field file), the third column is the atom designation (either **QM**, **MM**, or **LI** for a link atom), and the remaining entries are the atom indices to which the current atom is bonded in MM. Note that in the connection table, all **MM** atoms must come at the very end.

As the file content is copied verbatim, it is possible to add further sublocks to the QM/MM input section. This is necessary if any of the atoms is designated as link atom (label **LI**). In this case, the **LINK_BONDS** section needs to be added:

```
1       SO      QM      2
2       C       QM      1       3       4
3       HP      QM      2
4       CT      QM      2       5       6       7
5       H1      QM      4
6       H1      QM      4
7       CT      LI      4       8       9       10
8       HC      MM      7
9       HC      MM      7
10      HC      MM      7
  subend
  link_bonds
7 - 4  1.4  H.1  HC
```

Note how the **MM_CONNECTION_TABLE** block is terminated by **subend** and then the **LINK_BONDS** section starts; the **LINK_BONDS** section is not terminated by **subend** because the interface adds a **subend** after the file content. Within the **LINK_BONDS** section, each line has the following structure: (i) atom index of the **LI** atom; (ii) a **-** separated by spaces; (iii) atom index of a **QM** atom bonded to the link atom; (iv) a floating point number giving the parameter f; (v) the atomic fragment type of the atom which replaces the **LI** atom in the QM calculation; (vi) the MM atom type of the replacing atom. Note that for ADF the order of (i) and (iii) does not matter, but it does for the interface, and giving the atom index of the **QM** atom first will currently lead to an error. Given these parameters, internally ADF will replace the **LI** atom by an atom of the specified atomic fragment type, which will be placed on the **QM**–**LI** bond where the new bond length is the old bond length divided by f.

The file can also be used to override the MM charges for specific atoms:

```
1       SO      QM      2
2       C       QM      1       3       4
```

```
3       H1      QM      2
4       H1      QM      2
5       OW      MM      6      7
6       HW      MM      5
7       HW      MM      5
 subend
 charges
5  -0.96
6  +0.48
7  +0.48
```

The same rules regarding the **subend** as above apply.

6.7.5 Input file generator: ADF_input.py

In order to quickly setup simple calculations using ADF, the SHARC suite contains a small script called **ADF_input.py**. It can be used to setup single point calculations, optimizations, and frequency calculations on the DFT and TD-DFT level of theory. Of course, ADF has far more capabilities, but these are not covered by **ADF_input.py**. For more complicated inputs, users should manually adjust the generated input or employ **adfinput**.

Note that the script cannot write template files for **SHARC_ADF.py**. These templates can better be created by modifying the example template file in **$SHARC/../examples/SHARC_ADF/ADF.template**.

The script interactively asks the user to specify the calculation and writes an input file and optionally a run script.

Input

Type of calculation Choose to either perform a single-point calculation or a minimum optimization (including optionally frequency calculation).

The standard output or **TAPE21** files from a frequency calculation can be converted (see section ??) to a file in MOLDEN format, which can then be used to generate initial conditions with **wigner.py**.

Geometry, Charge, Multiplicity Specify the geometry file in xyz format. Number of atoms and total nuclear charge is detected automatically. After the user inputs the total charge, the number of electrons is calculated automatically. The number of unpaired electrons can be specified to define the multiplicity.

Basis set With **ADF_input.py** it is only possible to enter a single basis set for all atoms. More complicated basis setups can be achieved by manually adjusting the generated input file.

Level of theory One can setup calculations with any ADF-supported functional, and for ground state or excited state. The user is also queried whether relativistic effects need to be included, and for the most important accuracy settings.

6.7.6 Frequencies converter: ADF_freq.py

The small script **ADF_freq.py** can be used to convert the standard output or the **TAPE21** file created by an ADF frequency calculation. The usage is very simple:

```
$SHARC/ADF_freq.py ADF.out
```

or:

```
$SHARC/ADF_freq.py TAPE21
```

The script detects automatically the file format. Note that in ADF, the infrared intensities are only accessible from the standard output, so use this for IR spectrum generation. However, the data in **`TAPE21`** has a higher numeric precision, so it is recommended to convert the **`TAPE21`** file if no intensities are needed. In any case, a file called **`<filename>.molden`** is written, containing the frequencies and normal modes. This file can then be used with **`wigner.py`**.

Note that in SHARC2.1, this script was changed to use the **`KFFile`** library of ADF, meaning that (like **`SHARC_ADF.py`**, you will need ADF2019 or newer and Python 3.5 or higher to run it.

6.8 RICC2 Interface

The SHARC-RICC2 interface can be used to conduct excited-state dynamics based on TURBOMOLE's CC2 and ADC(2) methods, for singlet and triplet states. The interface uses the programs **`define`**, **`dscf`** and **`ricc2`**. For spin-orbit couplings, ORCA needs to be installed in addition to TURBOMOLE. Only ADC(2) can be used to calculate spin-orbit couplings, but not CC2 (hence, it is not recommended to perform CC2 calculations with both singlet and triplet states). Even with ADC(2), only singlet-triplet SOCs are obtained, but no triplet-triplet SOCs; S_0-triplet SOCs are also missing currently. THEODORE (version 2.0 and higher) can be used to perform automatic wave function analysis. Wavefunction overlaps are calculated using the WFOVERLAP code of the González group.[?]

The SHARC-RICC2 interface furthermore allows to perform QM/MM dynamics (ADC(2) or CC2 plus force fields), using TINKER for the MM part.

The interface needs two additional input files, a template file for the quantum chemistry (file name is **`RICC2.template`**) and a general input file (**`RICC2.resources`**). If a file **`QM/mos.init`** is are present, it is used to provide an initial orbital guess for the SCF calculation. In the case of QM/MM calculations, two more input files are needed: an AMBER95 force field file, and **`RICC2.qmmm.table`**, which contains the connectivity and force field IDs per atom.

6.8.1 Template file: `RICC2.template`

This file contains the specifications for the quantum chemistry calculation. Note that this is not a valid TURBOMOLE input file. The file only contains a number of keywords, given in table ??. The actual input for TURBOMOLE will be generated automatically through **`define`**. A fully commented template file with all possible options is located in **`$SHARC/../examples/SHARC_RICC2/`**.

External basis set libary

If users want to employ their own basis sets, they can create a basis set library directory with the required files, and use the **`basislib`** keyword to tell the interface to use this directory. The **`basislib`** keyword cannot be used together with the **`auxbasis`** keyword.

The specified directory must contain **`basen/`** and **`cbasen/`** subdirectories. These must contain one file per element, containing the desired basis set parameters. The files in **`cbasen/`** must auxiliary basis sets of the same name as the basis sets in **`basen/`**. See the TURBOMOLE directory structure to see how the directories and files need to be prepared.

6.8.2 Resource file: `RICC2.resources`

The file **`RICC2.resources`** contains mainly paths (to the TURBOMOLE and ORCA executables, to the scratch directory, etc.). This file must reside in the same directory where the interface is started. It uses a simple "**`keyword argument`**" syntax. Comments using # and blank lines are possible, the order of keywords is arbitrary. Lines with unknown keywords are ignored, since the interface just searches the file for certain keywords.

Table ?? lists the existing keywords. A fully commented resource file with all possible options is located in **`$SHARC/../examples/SHARC_R`**

Note that the interface sets all environment variables necessary to run TURBOMOLE (e.g., **`$PATH`**) automatically, based on the input from **`RICC2.resources`** and **`QM.in`**.

For parallel calculations, the interface will call the SMP executables of TURBOMOLE and WFOVERLAP.

Note that the **`dipolelevel`** keyword can have significant impact on the calculation time. Generally, in response methods like CC2 and ADC(2), extra computational effort is required for the calculation of state and transition dipole moments

Table 6.11: Keywords for the **RICC2.template** file.

Keyword	Description
`basis`	The basis set used. The interface will convert this string to the correct case for TURBOMOLE.
`auxbasis`	The auxiliary basis set used in **ricc2**. If no auxbasis is given, the interface will let **define** decide on a suitable auxbasis.
`basislib`	Path to external basis set library. Can be used to employ custom basis sets.
`charge`	The total molecular charge. The interface will calculate the number of electrons automatically during setup.
`method`	Followed by a string, which is either "CC2" or "ADC(2)" (case insensitive), defining the level of theory. Default is "ADC(2)".
`douglas-kroll`	Activates the use of the scalar-relativistic DK Hamiltonian of 2nd order. Default (if no keyword is given) is to use the non-relativistic Hamiltonian.
`spin-scaling`	Followed by a string, which is either "none", "scs" or "sos". Using these options, spin-component scaling can be activated. Under certain restrictions (no SOC, no transition dipole moments, no SMP), "lt-sos" can be used to perform cheaper "sos" calculations.
`scf`	Followed by a string which is either "dscf" or "ridft". Using this option, the SCF program can be chosen. Note that currently, there is no advantage of using "ridft".
`frozen`	Followed by an integer giving the number of frozen core orbitals in the **ricc2** calculations. Default is to use frozen core orbitals and let **define** decide on the number. If frozen core is not wanted, use **frozen 0** in the template.
`qmmm`	Activates QM/MM with electrostatic embedding.

. However, dipole moments have only influence in the dynamics simulations if a laser field is present. Using the **dipolelevel** keyword, it is possible to deactivate dipole moment calculations if they are not required. There are three different settings for **dipolelevel**:

- **dipolelevel**=0: The interface will return only dipole moments which can be calculated at no cost (state dipole moments of states where a gradient is calculated; excited-excited transition dipole moments if SOCs are calculated)
- **dipolelevel**=1: In addition, the interface will calculate transition dipole moments between S_0 and excited singlet states. Use at least this level for the initial condition setup (**setup_init.py** takes care of this).
- **dipolelevel**=2: The interface will calculate all state and transition dipole moments

If only energies and dipole moments are calculated, **dipolelevel**=1 is only slightly more expensive than **dipolelevel**=0, while **dipolelevel**=2 increases computation time more strongly. However, the computation time also depends on whether or not spin-orbit couplings and gradients are calculated.

6.8.3 QM/MM force field file

These force field files should have the format that can be found in the TINKER directory.

6.8.4 QM/MM connection table file: RICC2.qmmm.table

This file defines which atoms are in the QM or MM region, the atom types (for TINKER), and the connectivity. Note that the SHARC-RICC2 interface uses newly developed routines to setup QM/MM calculations and communicate with TINKER, so this file is not identical to the table files of the ADF or MOLCAS interfaces (it uses the same routines as the SHARC-ORCA interface).

A sample looks like:

```
QM    223
QM    222        1  3  4
QM    136
QM     80
MM     63        6 7
```

Table 6.12: Keywords for the **RICC2.resources** file.

Keyword	Description
turbodir	Is the path to the TURBOMOLE installation directory. This directory should contain subdirectories like **bin/**, **basen/**, **cbasen/**, or **scripts/**. Relative and absolute paths, environment variables and ~ can be used. The interface will set **$TURBODIR** to this path, and will set the **$PATH** correctly (using TURBOMOLE's **sysname** tool. If this keyword is not present in **RICC2.resources**, the interface will use the environment variable **$TURBODIR**, if it is set.
orcadir	Is the path to the ORCA installation, which is necessary when spin-orbit couplings are calculated. Relative and absolute paths, environment variables and ~ can be used. If this keyword is not present in **RICC2.resources**, the interface will use the environment variable **$ORCADIR**, if it is set. Note that it is possible that ORCA 4.1 or later will not be compatible with mkl files written by TURBOMOLE, so that an older ORCA version is required.
tinker	The path to the TINKER installation directory. This directory should contain the **bin/** subdirectory.
scratchdir	Is a path to the temporary directory. Relative and absolute paths, environment variables and ~ can be used. If it does not exist, the interface will create it. In any case, the interface will delete this directory after the calculation.
savedir	Is a path to another temporary directory. Relative and absolute paths, environment variables and ~ can be used. The interface will store files needed for restart there.
memory	The memory usable by TURBOMOLE and WFOVERLAP. The default is 100 MB.
ncpu	The number of CPUs used by the interface. If larger than one, the interface will use the SMP executables of TURBOMOLE with the given number of CPUs.
always_orb_init	Do not use the orbital guesses from previous calculations/time steps, but always use the provided initial orbitals.
always_guess	Always use the orbital guess from the EHT calculation of **define**.
dipolelevel	Followed by an integer which is either 0, 1, or 2. Controls which dipole moment calculations are skipped by the interface.
wfoverlap	Is the path to the WFOVERLAP executable. Relative and absolute paths, environment variables and ~ can be used. This keyword is only necessary if overlaps need to be calculated.
wfthres	Threshold for the truncation of the response vectors which are used in the overlap calculations. A threshold of x implies that only the minimal number of determinants needed to give a norm of $1 - x$ are included.
numfrozcore	Overrides the number of frozen core orbitals for the overlap calculation. Default is to use the frozen orbitals from the RICC2 steps.
nooverlap	Do not save files necessary to do overlaps.
debug	Increases the verbosity of the interface.
no_print	Reduces interface standard output.

```
MM     64
MM     64
```

Each line corresponds to one atom. The example has 7 atoms, the first four in the QM region and the three last ones in the MM region. The second column defines the atom types according to the numbering in the used force field file. The integers to the right define the connectivity. Note that the interface automatically adds any necessary redundancy to the connectivity table, so it is enough in the table file to define a bond once.

6.9 LVC Interface

The purpose of the LVC interface is to allow performing computationally efficient dynamics using a linear vibronic coupling (LVC) model [?]. The relevant equations can be found in section **??**.

6.9.1 Input files

Two input files are needed:

`V0.txt` Contains all information to describe V_0: the equilibrium geometry, the frequencies ω_i, and an orthogonal matrix containing the normal coordinates $K_{\alpha i}$.

```
Geometry
 S     16.    0.00000000    0.00000000   -0.00039079   31.97207180
 O      8.    0.00000000   -2.38362453    1.36159121   15.99491464
 O      8.    0.00000000    2.38362453    1.36159120   15.99491464
Frequencies
 0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.00234  0.0053  0.0064
Mass-weighted normal modes
  0.0000  1.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
  0.6564  0.0000  0.0000  0.3108  0.0494  0.2004  0.0000  0.0000 -0.6557
...
```

`V0.txt` can be created from a MOLDEN file using **`wigner.py`**.

`LVC.template` Contains the state-specific information: ϵ_i, $\kappa_i^{(n)}$, $\lambda_j^{(m,n)}$, η_{mn}, as well as (transition) dipole moments. Here, for multiplets ϵ_i, $\kappa_i^{(n)}$, and $\lambda_j^{(m,n)}$ are shared between the multiplet components, whereas in the SOC and dipole moment matrices the multiplet components need to be considered explicitly.

```
V0.txt
4 0 3
epsilon
7
  1   1  0.0000000000
  1   2  0.1640045037
  1   3  0.1781507393
  1   4  0.2500917150
  3   1  0.1341247878
  3   2  0.1645714941
  3   3  0.1700512159
kappa
12
```

```
  1   2     7  1.19647e-03
  1   2     8  1.21848e-02
...
lambda
3
  1   1   4     9 -1.83185e-02
  1   2   3     9  7.32022e-03
  3   1   3     9  5.71746e-03
SOC R
  0.000e+00  0.000e+00  0.000e+00  0.000e+00  0.000e+00  0.000e+00 ...
  0.000e+00  0.000e+00  0.000e+00  0.000e+00  0.000e+00 -1.086e-04 ...
...
SOC I
  0.000e+00  0.000e+00  0.000e+00  0.000e+00  0.000e+00  0.000e+00 ...
  0.000e+00  0.000e+00  0.000e+00  1.000e+00  0.000e+00  1.000e-04 ...
...
DMX R
 -7.400e-07 -1.639e-01  0.000e+00  3.000e-06  0.000e+00  0.000e+00  ...
 -1.639e-01  3.930e-06  0.000e+00  2.400e-05  0.000e+00  0.000e+00  ...
...
DMX I
...
DMY R
...
DMY I
...
DMZ R
...
DMZ I
...
```

Only the two first lines are mandatory, but then all states will have the same potentials (the ground state potential).

6.9.2 Template File Setup: `wigner.py`, `setup_LVCparam.py`, and `create_LVCparam.py`

Reference potential **`V0.txt`** is created using the **`wigner.py`** script, which is also used for initial condition generation. Simply call, e.g.

```
$SHARC/wigner.py -l <filename.molden>
```

Note that this best works if all $3N$ normal modes are present in the file. If the translations and rotations are missing, the script will add the necessary number of zero-vector modes.

Setup for single point calculations To obtain the LVC parameters, two steps are necessary: (i) running quantum chemistry calculations, and (ii) converting the quantum chemistry output to the LVC parameters.

The first of these steps is carried out with **`setup_LVCparam.py`**. It is an interactive script that works very similarly to the other setup scripts (e.g., **`setup_init.py`**, **`setup_traj.py`**). The script will ask for the following:

- Path to the **`V0.txt`** file,
- Number of states,
- Which interface to use (in principle, all SHARC-interfaces can be used, but only the ab initio interfaces are useful here),
- Whether spin-orbit couplings should be calculated (only if applicable),
- Whether κ parameters should be obtained from analytical gradients or numerical differentiation (depends on availability of analytical gradients),

- Whether λ parameters should be obtained from analytical nonadiabatic coupling vectors or numerical differentiation (depends on availability of analytical nonadiabatic coupling vectors),
- Which normal modes to include,
- Which displacement value to use for numerical differentiation (default 0.05 dimensionless mass-frequency scaled units),
- Whether intruder states should be ignored or not,
- Whether one- or two-sided differentiation should be done,
- Interface-specific input for the chosen quantum chemistry interface (see section ??).

The script will set up a directory (**`DSPL_RESULTS`**) with one subdirectory for each single point calculation. If both κ's and λ's are computed analytically, then only the **`DSPL_000_eq`** subdirectory will be present. Otherwise, for each chosen normal mode 1–2 subdirectories (for one-sided or two-sided differentiation) will be present. Additionally, **`displacements.log`** presents the most important settings. In all cases, **`displacements.json`** is also present; this file is crucial to communicate all settings to the read-out script after the single point jobs are finished.

Extracting LVC parameters After all jobs are successfully finished (**`QM.out`** file present in each directory), run **`create_LVCparam.py`** inside the **`DSPL_RESULTS`** directory. This is a fully automatic script that reads **`displacements.json`** and the **`QM.out`** files in the subdirectories. After everything is successfully read, it creates the **`LVC.template`** file. This file can be used to run SHARC-LVC trajectories.

6.10 GAUSSIAN Interface

The SHARC-GAUSSIAN interface allows to run SHARC simulations with GAUSSIAN's TD-DFT functionality. The interface is compatible with restricted and unrestricted ground states, but not with symmetry. Spin-orbit couplings cannot be computed, but wave function overlaps from the WFOVERLAP code are available (no nonadiabatic couplings). Dyson norms can also be computed through the WFOVERLAP code. THEODORE (version 2.0 or higher) can be used to perform automatic wave function analysis.

The interface needs two additional input files, a template file for the quantum chemistry (file name is **`GAUSSIAN.template`**) and a resource file (**`GAUSSIAN.resources`**). If files **`QM/GAUSSIAN.chk.init`** or **`QM/GAUSSIAN.chk.<job>.init`** are present, they are used to provide an initial orbital guess for the SCF calculation of the respective job.

6.10.1 Template file: `GAUSSIAN.template`

This file contains the specifications for the quantum chemistry calculation. Note that this is not a valid GAUSSIAN input file. The file only contains a number of keywords, given in table ??. The actual input for GAUSSIAN will be generated automatically through the interface.

A fully commented template file—with all possible options, a comprehensive descriptions, and some practical hints—is located in **`$SHARC/../examples/SHARC_GAUSSIAN/GAUSSIAN.template`**. We recommend that users start from this template file and modify it appropriately for their calculations.

6.10.2 Resource file: `GAUSSIAN.resources`

The file **`GAUSSIAN.resources`** contains mainly paths (to the GAUSSIAN executables, to the scratch directory, etc.) and other resources, plus settings for **`wfoverlap.x`** and THEODORE. This file must reside in the same directory where the interface is started. It uses a simple "**`keyword argument`**" syntax. Comments using # and blank lines are possible, the order of keywords is arbitrary. Lines with unknown keywords are ignored, since the interface just searches the file for certain keywords.

Table ?? lists the existing keywords. A fully commented resource file with all possible options and comprehensive descriptions is located in **`$SHARC/../examples/SHARC_GAUSSIAN/`**.

Parallelization GAUSSIAN usually shows very good parallel scaling for most TD-DFT calculations. However, it is more efficient to carry out multiple GAUSSIAN calculations (different multiplicities, multiple gradients) in parallel, each one using a smaller number of CPUs.

Table 6.13: Keywords for the **`GAUSSIAN.template`** file.

Keyword	Description
`basis`	Gives the basis set for all atoms (default 6-31G).
`functional`	followed by one string giving the exchange-correlation functional. Default is **`PBEPBE`**.
`dispersion`	Activates dispersion correction. Arguments are written verbatim to GAUSSIAN input (in **`EmpiricalDispersion=()`**). Default is no dispersion. An example argument would be **`GD3`**.
`charge`	Sets the total charge of the system. Can be either followed by a single integer (then the interface will automatically assign the charges to the multiplicities) or by one charge per multiplicity.
`scrf`	Activates solvation. All arguments (e.g., **`iefpcm solvent=water`**) are copied to GAUSSIAN input (in **`scrf=()`**).
`grid`	Followed by a string (e.g., **`grid finegrid`**) defining which integration grid and accuracy to use. For details, see the example template file.
`denfit`	Activates density fitting, which might speed up the computation.
`scf`	Arguments are written verbatim to GAUSSIAN input (in **`scf=()`**).
`no_tda`	This keyword deactivates TDA, which the interface requests by default.
`paddingstates`	Followed by a list of integers, which give the number of extra states to compute by GAUSSIAN, but which are neglected in the output. Should not be changed between time steps, as this will break GAUSSIAN's restart routines.
`unrestricted_triplets`	Requests that the triplets are calculated in a separate job from an unrestricted ground state. Default is to compute triplets as linear response of the restricted singlet ground state.
`iop`	Arguments are written verbatim to GAUSSIAN input (in **`iop=()`**). Expert option.
`keys`	Arguments are written verbatim to GAUSSIAN input as separate keywords. Expert option.

In the SHARC-GAUSSIAN interface, parallelization is controlled by the keywords **`ncpu`** and **`schedule_scaling`**. The first keyword controls the maximum number of CPUs which the interface is allowed to use for all GAUSSIAN runs simultaneously. The second keyword is the parallel fraction from Amdahl's Law, see Section **??**. With a value close to zero, the interface will try to run all jobs at the same time. With values close to one, jobs will be run sequentially with the maximum number of cores. Typical values for **`schedule_scaling`** are around 0.90 for both GGA functionals and hybrid functionals, possibly less for very small computations.

Table 6.14: Keywords for the **`GAUSSIAN.resources`** file.

Keyword	Description
`groot`	Is the path to the GAUSSIAN installation directory. This directory should contain the GAUSSIAN executables, e.g., **`g09`**/**`g16`**, **`l9999.exe`**, etc. Relative and absolute paths, environment variables and ~ can be used. The interface will set **`$GAUSS_EXEDIR`** to this path.
`scratchdir`	Is a path to the temporary directory. Relative and absolute paths, environment variables and ~ can be used. If it does not exist, the interface will create it. In any case, the interface will delete this directory after the calculation. The interface will set **`$GAUSS_SCRDIR`** to this path.
`savedir`	Is a path to another temporary directory. Relative and absolute paths, environment variables and ~ can be used. The interface will store files needed for restart there.
`memory`	The memory usable by GAUSSIAN and WFOVERLAP. The default is 100 MB.
`ncpu`	The number of CPUs used by the interface. Is overridden by environment variables from queuing engines (e.g., **`$NSLOTS`** or **`$SLURM_NTASKS_PER_NODE`**). Will either be used to run GAUSSIAN in parallel or to run several independent GAUSSIAN runs at the same time.
`schedule_scaling`	Gives the expected parallelizable fraction of the GAUSSIAN run time (Amdahl's law). With a value close to zero, the interface will try to run all jobs at the same time. With values close to one, jobs will be run sequentially with the maximum number of cores.
`delay`	Followed by a float giving the delay in seconds between starting parallel jobs to avoid excessive disk I/O (usually not necessary).
`wfoverlap`	Path to the WFOVERLAP code. Needed for overlap and Dyson norm calculations.
`wfthres`	(float) Gives the amount of wave function norm which will be kept in the truncation of the determinant files. Default is 0.99 (i.e., the wave functions in the determinant files will have a norm of 0.99). Note that if hybrid functionals and no TDA are used, the response vector can have a norm larger than one, and wfthres should be increased.
`numfrozcore`	Number of frozen core orbitals for overlap and Dyson norm calculations. A value of -1 enables automatic frozen core.
`numocc`	Number of ignored occupied orbitals in Dyson calculations.
`nooverlap`	Do not save determinant files for overlap computations.
`theodir`	Path to the THEODORE installation. Relative and absolute paths, environment variables and ~ can be used. The interface will set **`$PYTHONPATH`** automatically.
`theodore_prop`	Followed by a list with the descriptors which THEODORE should compute. Note that descriptors will only be computed for restricted singlets (and triplets). Instead of a simple list, a Python literal can also be used, as in the THEODORE input files.
`theodore_fragment`	Followed by a list of atom numbers which should constitute a fragment in THEODORE. For multiple fragments, the keyword can be used multiple times. Instead, the keyword can be followed by a Python literal, as in the THEODORE input files.
`always_orb_init`	Do not use the orbital guesses from previous calculations/time steps, but always use the provided initial orbitals.
`always_guess`	Always use the orbital guess from GAUSSIAN.
`debug`	Increases the verbosity of the interface (standard out). Does not clean up the scratch directory. Copies all GAUSSIAN outputs to the save directory.
`no_print`	Reduces interface standard output.

6.11 ORCA Interface

The SHARC-ORCA interface allows to run SHARC simulations with ORCA's TD-DFT functionality. The interface is compatible with restricted and unrestricted ground states, but not with symmetry. Spin-orbit couplings can be computed and wave function overlaps from the WFOVERLAP code are available (no nonadiabatic couplings). Dyson norms can also be computed through the WFOVERLAP code. THEODORE (version 2.0 or higher) can be used to perform automatic wave function analysis. Note that the interface only works with ORCA 4.1 and newer (you can check by looking whether the program **`orca_fragovl`** is present in the ORCA installation). Initial tests on the recently released ORCA 4.2 were successful, but if problems are encountered, users should try with ORCA 4.1 instead.

The SHARC-ORCA interface furthermore allows to perform QM/MM dynamics (TDDFT plus force fields), using TINKER for the MM part.

The interface needs two additional input files, a template file for the quantum chemistry (file name is **`ORCA.template`**) and a resource file (**`ORCA.resources`**). If files **`QM/ORCA.gbw.init`** or **`QM/ORCA.gbw.<job>.init`** are present, they are used to provide an initial orbital guess for the SCF calculation of the respective job. In the case of QM/MM calculations, two more input files are needed: an AMBER95 force field file, and **`ORCA.qmmm.table`**, which contains the connectivity and force field IDs per atom.

6.11.1 Template file: ORCA.template

This file contains the specifications for the quantum chemistry calculation. Note that this is not a valid ORCA input file. The file only contains a number of keywords, given in table ??. The actual input for ORCA will be generated automatically through the interface.

A fully commented template file—with all possible options, a comprehensive descriptions, and some practical hints—is located at **`$SHARC/../examples/SHARC_ORCA/ORCA.template`**. For QM/MM calculations, a second example template (along with the QM/MM-specific files) is located at **`$SHARC/../examples/SHARC_ORCA_Tinker/ORCA.template`**. We recommend that users start from this template file and modify it appropriately for their calculations.

Note that for ORCA 4.2, when doing TD-DFT gradient calculations with hybrid functions, it might be necessary to add a "/C" basis set. While there is no special keyword in **`ORCA.template`** for this basis set, it can be specified with the general **`key`** keyword.

6.11.2 Resource file: ORCA.resources

The file **`ORCA.resources`** contains mainly paths (to the ORCA executables, to the scratch directory, etc.) and other resources, plus settings for **`wfoverlap.x`** and THEODORE. This file must reside in the same directory where the interface is started. It uses a simple "**`keyword argument`**" syntax. Comments using # and blank lines are possible, the order of keywords is arbitrary. Lines with unknown keywords are ignored, since the interface just searches the file for certain keywords.

Table ?? lists the existing keywords. A fully commented resource file with all possible options and comprehensive descriptions is located in **`$SHARC/../examples/SHARC_ORCA/`**.

Parallelization ORCA usually shows very good parallel scaling for most TD-DFT calculations. However, it is more efficient to carry out multiple ORCA calculations (different multiplicities) in parallel, each one using a smaller number of CPUs. Note that ORCA 4.1 can compute multiple gradients in one job, so multiple gradients do not induce multiple jobs.

In the SHARC-ORCA interface, parallelization is controlled by the keywords **`ncpu`** and **`schedule_scaling`**. The first keyword controls the maximum number of CPUs which the interface is allowed to use for all ORCA runs simultaneously. The second keyword is the parallel fraction from Amdahl's Law, see Section ??. With a value close to zero, the interface will try to run all jobs at the same time. With values close to one, jobs will be run sequentially with the maximum number of cores. Typical values for **`schedule_scaling`** are around 0.90 for both GGA functionals and hybrid functionals, possibly less for very small computations.

6.11.3 QM/MM force field file

These force field files should have the format that can be found in the TINKER directory.

Table 6.15: Keywords for the **ORCA.template** file.

Keyword	Description
basis	Gives the basis set for all atoms (default 6-31G).
auxbasis	Gives the auxiliary basis set (default: Orca chooses).
basis_per_element	Overrides the basis set for the specified element (argument 1: element symbol, argument 2: basis set). Can be given multiple times.
basis_per_atom	Overrides the basis set for the specified atom (argument 1: atom number starting at 1 and only counting QM atoms, argument 2: basis set). Can be given multiple times.
functional	followed by one string giving the exchange-correlation functional. Default is **PBE**.
hfexchange	Modifies the amount of HF exchange in the functional (give in fraction of 1). Default is whatever the chosen functional uses.
dispersion	Activates dispersion correction. Arguments are written verbatim to ORCA input (in **EmpiricalDispersion=()**). Default is no dispersion. An example argument would be **GD3**.
range_sep_settings	Controls several functional parameters. Is followed by 5 arguments: **RangeSepMu**, **RangeSepScal**, **ACM-A**, **ACM-B**, **ACM-C**. Each is a floating point number. See the ORCA manual for details on these settings. Note that **ACM-A** should not be used together with **hfexchange**.
charge	Sets the total charge of the system. Can be either followed by a single integer (then the interface will automatically assign the charges to the multiplicities) or by one charge per multiplicity.
grid	Followed by a number (e.g., **grid 4**) defining which integration grid and accuracy to use. For details, see the example template file.
gridx	To control the exchange grid (**rijcosx**).
gridxc	To control the exchange-correlation grid (TDDFT).
ri	Controls the density fitting scheme. Arguments can be, e.g., **rijcosx**. If not given, RI is deactivated.
keys	Arguments are written verbatim to ORCA input as separate keywords. Expert option. Note that this can be used to do PCM calculations (but requesting gradients will lead to a crash).
no_tda	This keyword deactivates TDA, which the interface requests by default.
paddingstates	Followed by a list of integers, which give the number of extra states to compute by GAUSSIAN, but which are neglected in the output. Should not be changed between time steps, as this will break GAUSSIAN's restart routines.
unrestricted_triplets	Requests that the triplets are calculated in a separate job from an unrestricted ground state. Default is to compute triplets as linear response of the restricted singlet ground state.
qmmm	Activates QM/MM. In this case, the interface will look for two additional input files (see below).

Table 6.16: Keywords for the **ORCA.resources** file.

Keyword	Description
orcadir	Is the path to the ORCA installation directory. This directory should contain executables like **orca**, **orca_int**, or **orca_fragovl**. Relative and absolute paths, environment variables and ~ can be used. The interface will automatically update the **$LD_LIBRARY_PATH**.
tinker	The path to the TINKER installation directory. This directory should contain the **bin/** subdirectory.
scratchdir	Is a path to the temporary directory. Relative and absolute paths, environment variables and ~ can be used. If it does not exist, the interface will create it. In any case, the interface will delete this directory after the calculation.
savedir	Is a path to another temporary directory. Relative and absolute paths, environment variables and ~ can be used. The interface will store files needed for restart there.
memory	The memory usable by ORCA and WFOVERLAP. The default is 100 MB.
ncpu	The number of CPUs used by the interface. Is overridden by environment variables from queuing engines (e.g., **$NSLOTS** or **$SLURM_NTASKS_PER_NODE**). Will either be used to run ORCA in parallel or to run several independent ORCA runs at the same time.
schedule_scaling	Gives the expected parallelizable fraction of the ORCA run time (Amdahl's law). With a value close to zero, the interface will try to run all jobs at the same time. With values close to one, jobs will be run sequentially with the maximum number of cores.
delay	Followed by a float giving the delay in seconds between starting parallel jobs to avoid excessive disk I/O (usually not necessary).
wfoverlap	Path to the WFOVERLAP code. Needed for overlap and Dyson norm calculations.
wfthres	(float) Gives the amount of wave function norm which will be kept in the truncation of the determinant files. Default is 0.99 (i.e., the wave functions in the determinant files will have a norm of 0.99). Note that if hybrid functionals and no TDA are used, the response vector can have a norm larger than one, and wfthres should be increased.
numfrozcore	Number of frozen core orbitals for overlap and Dyson norm calculations. A value of -1 enables automatic frozen core.
numocc	Number of ignored occupied orbitals in Dyson calculations.
nooverlap	Do not save determinant files for overlap computations.
theodir	Path to the THEODORE installation. Relative and absolute paths, environment variables and ~ can be used. The interface will set **$PYTHONPATH** automatically.
theodore_prop	Followed by a list with the descriptors which THEODORE should compute. Note that descriptors will only be computed for restricted singlets (and triplets). Instead of a simple list, a Python literal can also be used, as in the THEODORE input files.
theodore_fragment	Followed by a list of atom numbers which should constitute a fragment in THEODORE. For multiple fragments, the keyword can be used multiple times. Instead, the keyword can be followed by a Python literal, as in the THEODORE input files.
always_orb_init	Do not use the orbital guesses from previous calculations/time steps, but always use the provided initial orbitals.
always_guess	Always use the orbital guess from ORCA.
debug	Increases the verbosity of the interface (standard out). Does not clean up the scratch directory. Copies all ORCA outputs to the save directory.
no_print	Reduces interface standard output.
qmmm_table	Followed by the path to the connection table file, in SHARC-QM/MM format.
qmmm_ff_file	Followed by the path to the force field file, in AMBER95 format for TINKER.
neglected_gradient	Decides how not-requested gradients are reported back (Options: **zero**: default, not-requested gradients are zero; **gs**: not-requested gradients are equal to ground state gradient; **closest**: not-requested gradients are equal to closest-energy requested gradient).

6.11.4 QM/MM connection table file: `ORCA.qmmm.table`

This file defines which atoms are in the QM or MM region, the atom types (for TINKER), and the connectivity. Note that the SHARC-ORCA interface uses newly developed routines to setup QM/MM calculations and communicate with TINKER, so this file is not identical to the table files of the ADF or MOLCAS interfaces.

A sample looks like:

```
QM    223
QM    222        1  3  4
QM    136
QM     80
MM     63        6 7
MM     64
MM     64
```

Each line corresponds to one atom. The example has 7 atoms, the first four in the QM region and the three last ones in the MM region. The second column defines the atom types according to the numbering in the used force field file. The integers to the right define the connectivity. Note that the interface automatically adds any necessary redundancy to the connectivity table, so it is enough in the table file to define a bond once.

6.12 BAGEL Interface

The SHARC-BAGEL interface allows to run SHARC simulations with BAGEL's CASSCF, SS-CASPT2, MS-CASPT2, and XMS-CASPT2 functionalities. The interface is compatible with all multiplicities, but not with symmetry. Note that separate active spaces are used for each multiplicity. BAGEL features analytical gradients and nonadiabatic couplings for all of these methods, but no spin-orbit coupings. Wave function overlaps and Dyson norms can be obtained from the WFOVERLAP code.

Note that BAGEL does not allow extracting the AO overlap matrix that is required for overlap and Dyson norm calculations; hence, the SHARC-BAGEL interface computes the AO overlaps through the open-source quantum chemistry code PYQUANTE. Also note that, currently, it is not recommended to work with MS-CASPT2 gradients and XMS-CASPT2 should always be preferred.

The interface needs two additional input files, a template file for the quantum chemistry (file name is **`BAGEL.template`**) and a resource file (**`BAGEL.resources`**). If files **`QM/archive.<mult>.init`** are present, they are used to provide an initial orbital guess for the CASSCF calculation of the respective multiplicity.

Note that when running BAGEL, it might be necessary for the user to set **`$LD_LIBRARY_PATH`** appropriately, so that all relevant libraries (**`boost`**, **`fabric`**, ...) can be found.

6.12.1 Template file: `BAGEL.template`

This file contains the specifications for the quantum chemistry calculation. Note that this is not a valid BAGEL input file. The file only contains a number of keywords, given in table ??. The actual input for BAGEL will be generated automatically through the interface.

A fully commented template file—with all possible options, a comprehensive descriptions, and some practical hints—is located at **`$SHARC/../examples/SHARC_BAGEL/BAGEL.template`**. For QM/MM calculations, a second example template (along with the QM/MM-specific files) is located at **`$SHARC/../examples/SHARC_BAGEL_Tinker/BAGEL.template`**. We recommend that users start from this template file and modify it appropriately for their calculations.

6.12.2 Resource file: `BAGEL.resources`

The file **`BAGEL.resources`** contains mainly paths (to the BAGEL executables, to the scratch directory, etc.) and other resources, plus settings relevant for **`wfoverlap.x`**. This file must reside in the same directory where the interface is started. It uses a simple "**`keyword argument`**" syntax. Comments using # and blank lines are possible, the order of

Table 6.17: Keywords for the **BAGEL.template** file.

Keyword	Description
`basis`	Gives the basis set for all atoms (default svp). It is advisable to always specify a basis set file with absolute path.
`df_basis`	Gives the auxiliary basis set (default: svp-jkfit). It is advisable to always specify a DF basis set file with absolute path.
`dkh`	Activates the (scalar-relativistic) Douglas-Kroll-Hess Hamiltonian.
`nact`	Number of active orbitals.
`nclosed`	Number of closed-shell orbitals.
`nstate`	Number of state-averaging states per multiplicity.
`method`	Can be **casscf**, **caspt2**, **ms-caspt2**, or **xms-caspt2**.
`shift`	Level shift for CASPT2 (default: 0.0, give float in Hartree).
`shift_imag`	Switches from real to imaginary level shift (use **shift** to specify the magnitude).
`orthogonal_basis`	Switches on the **orthogonal_basis** option of BAGEL. Is activated automatically for imaginary level shifts.
`msmr`	Switches on multi-state-multi-reference treatment in CASPT2. Default is single-state-single-reference (SS-SR).
`maxiter`	Iteration limit for energy calculations (SCF, PT2). Default 500. A too high value can slow down the calculation unnecessarily.
`maxziter`	Iteration limit for Z-vector calculations (gradients, NACME). Default 100. A too high value can slow down the calculation unnecessarily.
`charge`	Sets the total charge of the system. Can be either followed by a single integer (then the interface will automatically assign the charges to the multiplicities) or by one charge per multiplicity.
`frozen`	Number of frozen core orbitals for CASPT2 steps. Default is -1, which lets BAGEL automatically decide.

keywords is arbitrary. Lines with unknown keywords are ignored, since the interface just searches the file for certain keywords.

Table **??** lists the existing keywords. A fully commented resource file with all possible options and comprehensive descriptions is located in **$SHARC/../examples/SHARC_BAGEL/**.

Note that the **dipolelevel** keyword can have significant impact on the calculation time. Generally, in CASPT2 calculations, extra computational effort is required for the calculation of state and transition dipole moments . However, dipole moments have only influence in the dynamics simulations if a laser field is present. Using the **dipolelevel** keyword, it is possible to deactivate dipole moment calculations if they are not required. There are three different settings for **dipolelevel**:

- **dipolelevel**=0: The interface will return only dipole moments which can be calculated at no cost (state dipole moments of states where a gradient is calculated; transition dipole moments if nonadiabatic couplings are calculated)
- **dipolelevel**=1: In addition, the interface will calculate transition dipole moments between S_0 and excited singlet states. Use at least this level for the initial condition setup (**setup_init.py** takes care of this).
- **dipolelevel**=2: The interface will calculate all state and transition dipole moments

If only energies and dipole moments are calculated, **dipolelevel**=1 is only slightly more expensive than **dipolelevel**=0, while **dipolelevel**=2 increases computation time more strongly. However, the computation time also depends on whether or not nonadiabatic couplings and gradients are calculated.

Parallelization The parallel scaling behavior of BAGEL heavily depends on the system (number of atoms, active space, frozen core, ...) and on the parallelization mode (MPI or OpenMP). Hence, it is advisable that the optimal settings (number of cores, parallelization mode) are tested before starting dynamics projects. Note that the interface can only trivially parallelize BAGEL calculations across several independent multiplicities, but not across multiple gradient or nonadiabatic coupling calculations.

Table 6.18: Keywords for the **`BAGEL.resources`** file.

Keyword	Description
`bagel`	Is the path to the BAGEL installation directory. This directory should contain subdirectories **`bin/`** and **`lib/`**. Relative and absolute paths, environment variables and ~ can be used. The interface will automatically update the **`$LD_LIBRARY_PATH`**.
`scratchdir`	Is a path to the temporary directory. Relative and absolute paths, environment variables and ~ can be used. If it does not exist, the interface will create it. In any case, the interface will delete this directory after the calculation.
`savedir`	Is a path to another temporary directory. Relative and absolute paths, environment variables and ~ can be used. The interface will store files needed for restart there.
`pyquante`	Path to the PYQUANTE installation directory. This directory should contain the **`PyQuante/`** subdirectory that can be imported by Python as a module. Required for overlap and Dyson calculations.
`memory`	The memory usable by WFOVERLAP. The default is 100 MB. Note that the memory for BAGEL cannot be controlled.
`ncpu`	The number of CPUs used by the interface. Is overridden by environment variables from queuing engines (e.g., **`$NSLOTS`** or **`$SLURM_NTASKS_PER_NODE`**). Will either be used to run BAGEL in parallel or to run several independent BAGEL runs at the same time.
`schedule_scaling`	Gives the expected parallelizable fraction of the BAGEL run time (Amdahl's law). With a value close to zero, the interface will try to run all jobs at the same time. With values close to one, jobs will be run sequentially with the maximum number of cores.
`mpi_parallel`	If given, the interface will call BAGEL with **`mpirun -n NCPU`**, otherwise it will use OpenMP.
`delay`	Followed by a float giving the delay in seconds between starting parallel jobs to avoid excessive disk I/O (usually not necessary).
`wfoverlap`	Path to the WFOVERLAP code. Needed for overlap and Dyson norm calculations.
`numfrozcore`	Number of frozen core orbitals for overlap and Dyson norm calculations. A value of -1 enables automatic frozen core.
`numocc`	Number of ignored occupied orbitals in Dyson calculations.
`nooverlap`	Do not save determinant files for overlap computations.
`dipolelevel`	Followed by an integer which is either 0, 1, or 2. Controls which dipole moment calculations are skipped by the interface.
`always_orb_init`	Do not use the orbital guesses from previous calculations/time steps, but always use the provided initial orbitals.
`always_guess`	Always use the orbital guess from BAGEL.
`debug`	Increases the verbosity of the interface (standard out). Does not clean up the scratch directory. Copies all BAGEL outputs to the save directory.
`no_print`	Reduces interface standard output.

6.13 The WFOVERLAP Program

This section does not describe an interface to SHARC, but rather the WFOVERLAP program. This program is part of the SHARC distribution, but can also be obtained as a stand-alone packagehttps://sharc-md.org/?page$_id$ = 309$stand - alonepackage$ (including a more detailed manual and a set of auxiliary scripts). It computes overlaps between many-electron wave functions expressed in terms of linear combinations of Slater determinant, which are based on molecular orbitals (from an LCAO ansatz). It can also compute Dyson orbitals and Dyson norms between wave functions differing by one α or one β electron. The program is based on the efficient and general algorithm published in Ref. [?]. It is possible to vary the geometry, the basis set, the molecular orbitals, and the wavefunction expansion between the calculations.

The resulting wave function overlaps or Dyson norms can be used for example for:

- Propagation in local diabatization, the main application inside SHARC,
- Computation of photoionization spectra [? ?],
- Comparison of wave functions at different levels of theory [?].

If you employ the **`wfoverlap.x`** code inside the SHARC suite for these purposes, please cite these references!

The documentation here only gives a brief overview over the input options of WFOVERLAP.X, because within the SHARC suite the **`wfoverlap.x`** program is always called automatically by the interfaces. For the full manual (and for access to the auxiliary scripts of WFOVERLAP.X), please download the separate WFOVERLAP packagehttps://sharc-md.org/?page$_id$ = 309WFOVERLAP$package$.

6.13.1 Installation

Using precompiled binaries After unpacking, the directory **`$SHARC`** should contain a binary **`wfoverlap_ascii.x`** and a link called **`wfoverlap.x`** pointing to the binary. With this setup, most interfaces should work without problems.

Manual installation The only exceptions are the following: COLUMBUS (overlaps and Dyson norms) and MOLCAS (only Dyson norms). These features are only available if **`wfoverlap.x`** is recompiled with proper support for these programs. Alternatively, you may want to link **`wfoverlap.x`** against your favorite libraries. In these cases, a manual installation is necessary.

For the manual installation you need a working Fortran90 compatible compiler (Intel's ifort is recommended), some reasonably fast BLAS/LAPACK libraries (Intel's MKL is recommended, although atlas is also fine).

Optionally, with a working COLUMBUS Installation you can install the COLUMBUS bindings, which will allow direct reading of SIFS integral files generated by DALTON. To use this option, it is necessary to use the `read_dalton.o` object file. MOLCAS/SEWARD integral files can be read by linking with the COLUMBUS/MOLCAS interface. Link against `read_molcas.o` for this purpose.

To compile the source code, switch to the `source` directory and edit the Makefile to adjust it to your Fortran compiler and BLAS/LAPACK installation. The location of your COLUMBUS installation has to be set via the enviroment variable `$COLUMBUS`.

Issuing the command:

```
cd $SHARC/../wfoverlap/source/
make
```

will compile the source and create the binaries.

If you are unable to link against COLUMBUS and/or MOLCAS, simply call

```
make  wfoverlap_ascii.x
```

to compile a minimal version of the CI Overlap program that only reads ASCII files. In this case, overlap and Dyson calculations with **`SHARC_COLUMBUS.py`** and Dyson calculations with **`SHARC_MOLCAS.py`** will not be possible.

Testing The command

```
make test
```

will run a couple of tests to check if the program is working correctly (alternatively you can call `ovl_test.bash $OVDIR`, but **`$OVDIR`** needs to be set before).

6.13.2 Workflow

The workflow of the overlap program is shown in Figure **??**. Four pieces of input, as shown on top, have to be given:

- Overlaps between the two sets of AOs used to construct the bra and ket wavefunctions,
- MO coefficients of the bra and ket wavefunctions,
- information about the Slater determinants,
- the corresponding CI coefficients.

Two main intermediates are computed, the MO overlaps and the unique factors $\mathcal{S}_{kl}, \bar{\mathcal{S}}_{kl}$ where the latter may require significant amounts of memory to be stored. The reuse of these intermediates is one of the main reasons for the decent performance of the **`wfoverlap.`** program.

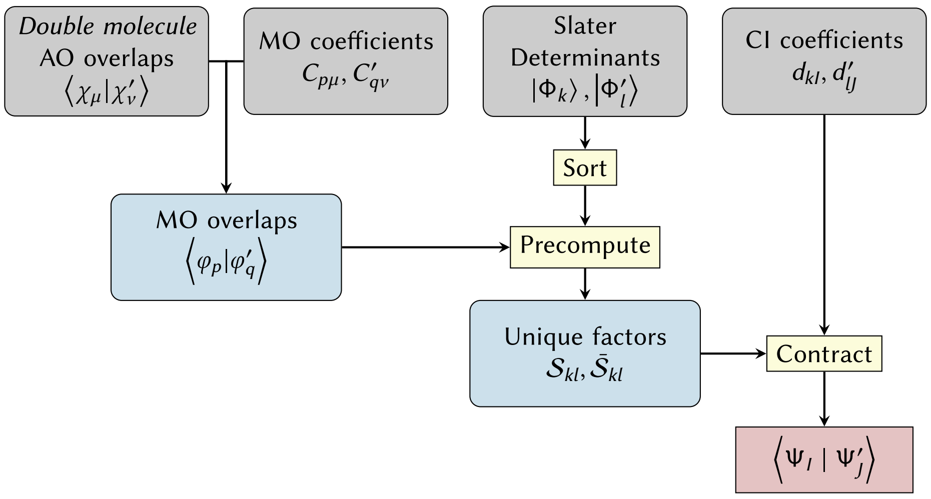
Double molecule
AO overlaps
$\langle \chi_\mu | \chi'_\nu \rangle$
MO coefficients
$C_{p\mu}, C'_{q\nu}$
Slater Determinants
$|\Phi_k\rangle, |\Phi'_l\rangle$
CI coefficients
d_{kI}, d'_{lJ}
Sort
MO overlaps
$\langle \varphi_p | \varphi'_q \rangle$
Precompute
Unique factors
$\mathcal{S}_{kl}, \bar{\mathcal{S}}_{kl}$
Contract
$\langle \Psi_I \mid \Psi'_J \rangle$

Figure 6.3: Workflow of the wavefunction overlap program.

6.13.3 Calling the program

The main program is called in the following form

```
wfoverlap.x [-m <mem=1000>] [-f <input_file=cioverlap.input>]
```

with the command line options

- **`-m`** : amount of memory in MB
- **`-f`** : input file

Example:

```
wfoverlap.x -m 2000 -f wfov.in
```

Mode The program automatically detects whether overlaps or Dyson orbitals should be calculated. If the number of electrons in the bra and ket wavefunctions is the same, wavefunction overlaps are computed. If the number of α electrons or the number of β electrons differ by exactly 1, Dyson orbitals are computed. If the wave functions differ by more than one electron, the program will stop with an error message.

Memory The amount of **memory** given is a decisive factor for the performance of the code. Depending on the amount of memory, one of three different modes is chosen:

(i) All $\mathcal{S}_{kl}$ and $\bar{\mathcal{S}}_{kl}$ terms are kept in core (using arrays called **P_ovl**and **Q_ovl**).

(ii) Only the $\mathcal{S}_{kl}$ factors (**P_ovl**) are kept in core. This is indicated by

```
Allocation of Q block overlap matrix failed.
- Using on-the-fly algorithm for Q determinants.
```

This mode is generally as efficient as 1. but shows somewhat worse parallel scaling.

(iii) Not even all $\mathcal{S}_{kl}$ factors can be stored

```
Only 437 out of 886 columns of the P_ovl matrix are stored in memory (3 MB)!
Increase the amount of memory to improve the efficiency.
```

This mode is significantly slower than (i) and (ii) and should be avoided by increasing the amount of memory.

Input file An example input file is shown below:

```
a_mo=mocoef_a
b_mo=mocoef_b
```

Table 6.19: List of keywords given in the input file. The **a_mo, b_mo, a_det, b_det** keywords are mandatory, all others are optional.

Keyword	Default	Description
a_mo	—	MO coefficient file (bra)
b_mo	—	MO coefficient file (ket)
a_mo_read	0	Format for the MO coefficients (bra): 0: COLUMBUS, 1: MOLCAS, 2: TURBOMOLE
b_mo_read	0	Format for the MO coefficients (ket)
a_det	—	Determinant file (bra)
b_det	—	Determinant file (ket)
ncore	0	Number of discarded core orbitals
ndocc	0	Number of doubly occupied orbitals that are not used for annihilation in Dyson orbital calculations (only has effect if larger than **ncore**)
ao_read	0	Format for overlap integrals: 0: ASCII, 1: MOLCAS, 2: COLUMBUS/SIFS, -1: Compute by inversion of MO coefficient matrx
mix_aoovl	**S_mix/ONEINT/aoints** for **ao_read=0/1/2**	AO overlap file
same_aos	.false.	If both calculations were performed with the same set of AOs (specify only for **ao_read=1/2**)
nao_a	*automatic*	Number of bra AOs for **ao_read=1/2** (specify only if different from ket AOs)
nao_b	*automatic*	Number of ket AOs (see above)
moprint	0	Print Dyson orbitals: 1: coefficients to std. out, 2: as Jmol script
force_direct_dets	.false.	Compute $\mathcal{S}_{kl}$ terms directly (turn off "superblocks"). Recommended if the number of CPU-cores is large (on the same order as the number of "superblocks").
force_noprecalc	.false.	Do not precalculate the $\bar{\mathcal{S}}_{kl}$ factors.
mixing_angles	.false.	Compute mixing angles as a matrix logarithm.

```
a_det=dets_a
b_det=dets_b
ao_read=2
same_aos
```

The full list of keywords is given in Table ??.

6.13.4 Input data

Typically, three types of input need to be provided: AO overlaps, MO coefficients, and a combined file with determinant information and CI coefficients (cf. Figure ??). The file formats are explained here. Within SHARC, these files are automatically extracted or converted by the interfaces, so the user does not need to create them.

AO overlaps The mixed AO overlaps $\langle \chi_\mu | \chi'_\nu \rangle$ between the AOs used to expand the bra and ket wavefunctions are required. They are in general created by a "double molecule" calculation, i.e. an AO integral calculation where every atom is found twice in the input file.

The native format (**`ao_read=0`**) is a simple ASCII file containing the relevant off-diagonal block of the mixed AO overlap matrix, e.g.

```
   7 7
     9.97108464676133E-001    2.36720699813181E-001   ...
     2.36720699813181E-001    9.99919192433940E-001   ...
     1.00147985713321E-002    6.52340422397770E-003   ...
         ...
```

In addition, MOLCAS (**`ao_read=1`**) and COLUMBUS/SIFS (**`ao_read=2`**) files can be read in binary form.

If the same AOs are used for the bra and ket wavefunctions and the MO coefficient matrix is square, it is possible to reconstruct the overlaps by inversion of the MO coefficient matrix (**`ao_read=-1`**). In this case it is not necessary to supply a **`mix_aoovl`** file.

MO coefficients MO coefficients of the bra and ket wavefunctions can usually be read in directly in the form written by the quantum chemistry program. The supported options for **`a_mo_read`** and **`b_mo_read`** are **`0`** for COLUMBUS format, **`1`** for MOLCAS lumorb format, and **`2`** for TURBOMOLE format.

Because the number of electrons strongly affects the run time of **`wfoverlap.x`**, it is generally beneficial to apply a frozen core approximation, even if the actual wave function calculation did not do so. Most interfaces which use **`wfoverlap.x`** have a keyword **`numfrozcore`** in the resource file, which only affects the number of frozen core orbitals for the overlap calculation (If the interface support frozen core for the quantum chemistry itself, there will be a keyword in the template file).

Slater determinants and CI coefficients Slater determinants and CI coefficients are currently supplied by an ASCII file of the form

```
3 7 168
dddddee   0.979083342437    0.979083342437   -0.122637656388
ddddabe  -0.094807515471   -0.094807515471   -0.663224542162
ddddbae   0.094807515471    0.094807515471    0.663224542162
...
```

The first line specifies

- the number of states (columns in the file),
- the number of MOs (length of the determinant strings), and
- the number of determinants in the file (length of the file).

Every subsequent line gives the determinant string and the corresponding CI coefficients for the different states. The following symbols are used in the determinant string:

d	- doubly occupied
a	- singly occupied (α)
b	- singly occupied (β)
e	- empty

Most relevant for SHARC users, the **wfoverlap.x** program *fully considers all determinants inside these files*, without applying any form of truncation. Hence, truncation of long wave functions is done during the creation of the determinant files. Most interfaces which write these files have a keyword **wfthres** in their resource file. This threshold is a number between 0.0 and 1.0, and is the minimum wave function norm to which the wave functions should be truncated. During truncation, the interfaces generally retain the largest-amplitude CI coefficients, and remove determinants with small coefficients, i.e., they find the truncated expansion with the fewest determinants which has a norm above the **wfthres**. Choosing this threshold properly can very strongly affect the computational time spent in the overlap calculation. Generally, for CASSCF wave functions the threshold can be set to 1 (**SHARC_MOLPRO.py**, **SHARC_MOLCAS.py**, and **SHARC_BAGEL.py** always use all determinants and do not have the **wfthres** option), for TDA-DFT/ADC(2) it should usually be well above 0.99, for and for MRCI wave functions it might be necessary to go as low as 0.95, depending on the accuracy and performance needed. For TD-DFT calculations without the Tamm-Damcoff approximation, the response vectors are usually normalized to $|\mathbf{X}|^2 - |\mathbf{Y}| = 1$, but only $\mathbf{X}$ is used in the overlap calculation; since the norm of $\mathbf{X}$ can thus exceed 1, the **wfthres** should be increased above 1, too. As a rule of thumb, the threshold should always be chosen such that each state is represented by at least a few 100 determinants in the file, in order to obtain smoothly varying overlaps. If unsure, the user should perform a test calculation, varying the **wfthres** until a suitable one is found (i.e., with as many determinants as possible such that the cost of the overlap calculation is bearable).

6.13.5 Output

Usually, the output of **wfoverlap.x** is automatically extracted by the interfaces, and reported in **QM.out** in the overlap or 2D-property sections.

Wavefunction overlaps The output first lists some information about the wavefunction structure and about the computational time taken for the individual steps (cf. Figure ??).

A typical result of a wavefunction overlap computation is shown here:

```
 Overlap matrix <PsiA_i|PsiB_j>
                |PsiB  1>     |PsiB  2>
<PsiA  1|    0.5162656622 -0.2040109070
<PsiA  2|   -0.2167057391 -0.5266552021

 Renormalized overlap matrix <PsiA_i|PsiB_j>
                |PsiB  1>     |PsiB  2>
<PsiA  1|    0.5162656622 -0.2040109070
<PsiA  2|   -0.2167057391 -0.5266552021

 Performing Lowdin orthonormalization by SVD...

 Orthonormalized overlap matrix <PsiA_i|PsiB_j>
                |PsiB  1>     |PsiB  2>
<PsiA  1|    0.9273847015 -0.3741090956
<PsiA  2|   -0.3741090956 -0.9273847015
```

Overlap matrix gives the raw overlap values

$$\langle\Psi_I|\Psi'_J\rangle \tag{6.7}$$

of the wavefunctions supplied.

`Renormalized overlap matrix` gives the renormalized overlap values

$$\frac{\langle\Psi_I|\Psi'_J\rangle}{||\Psi_I||^2||\Psi'_J||^2} \tag{6.8}$$

relevant in the case of wavefunction truncation.

The **`Orthonormalized overlap matrix`** is constructed according to a procedure described in more detail in Ref. [**?**].

Dyson orbitals The matrix of Dyson norms is printed at the end of the file

```
 ALPHA ionization
 Dyson norm matrix |<PsiA_i|PsiB_j>|^2
                |PsiB  1>     |PsiB  2>     |PsiB  3>
<PsiA  1|    0.8817323437  0.4716319904  0.0680618001
<PsiA  2|    0.0615587916  0.4657174978  0.8772909828
<PsiA  3|    0.0000000000  0.0363130811  0.0000000000
<PsiA  4|    0.9634885049  0.0000000000  0.0017379586
<PsiA  5|    0.0000000000  0.9261839484  0.0000000000
```

In the case of **`moprint=1`** the orbitals are printed, as well. The expansion is given with respect to the MOs of the neutral system.

```
  Dyson orbitals in reference |ket> MO basis:
<PsiA  1|
         |PsiB   1>      |PsiB   2>      |PsiB   3>
MO     1 -1.24032037E-03  0.00000000E+00  9.98160731E-04
MO     2 -5.90277699E-02  0.00000000E+00  6.14517859E-02
MO     3 -1.23295110E-09  0.00000000E+00  3.08416849E-10
MO     4  0.00000000E+00 -6.86713110E-01  0.00000000E+00
MO     5  9.31351013E-01  0.00000000E+00 -2.49427166E-01
MO     6 -3.50106110E-02  0.00000000E+00  4.19935829E-02
MO     7 -1.83303166E-10  0.00000000E+00 -3.67409953E-12
MO     8 -1.91183349E-10  0.00000000E+00  9.40768334E-11
...
```

7 Auxilliary Scripts

In this chapter, all auxiliary scripts and programs are documented. Input generators (like **molpro_input.py** and **molcas_input.py**) are documented in the relevant interface sections.

All auxiliary scripts are either interactive—prompting user input from stdin in order to setup a certain task—or non-interactive, meaning they are controlled by command-line arguments and options, in the same way as many standard command-line tools work.

All interactive scripts sequentially ask a number of questions to the user. In many cases, a default value is presented, which is either preset or detected by the scripts based on the availability of certain files. Furthermore, the scripts feature auto-completion of paths and filenames (use TAB), which is active only in questions where auto-completion is relevant. For certain questions where lists of integers needs to be entered, ranges can be indicated with the tilde symbol (~), e.g., **-8~-2** (note that no spaces are allowed between the tilde and the two numbers) to indicate the list **-8 -7 -6 -5 -4 -3 -2**.

All interactive scripts write a file called **KEYSTROKES.<script_name>** which contains the user input from the last completed session. These files can be piped to the interactive scripts to perform the same task again, for example:

```
user@host> cat KEYSTROKES.excite - | $SHARC/excite.py
```

Note the **-**, which tells **cat** to switch to stdin after the file ends, so that the user can proceed if the script asks for more input than contained in the **KEYSTROKES** file.

All non-interactive scripts can be called with the **-h** option to obtain a short description, usage information and a list of the command line options. Non-interactive scripts also write a **KEYSTROKES.<script_name>** file, which will contain the last command entered to execute the script (including all options and arguments).

All scripts can be safely killed during a run by using **Ctrl-C**. In the case of interactive scripts, a **KEYSTROKES.tmp** file remains, containing the user input made so far. Note that the **KEYSTROKES.tmp** file cannot be directly piped to the scripts, because **KEYSTROKES.tmp** will be overwritten when the script starts.

7.1 Wigner Distribution Sampling: wigner.py

The first step in preparing the dynamics calculation is to obtain a set of physically reasonable initial conditions. Each initial condition is a set of initial atomic coordinates, initial atomic velocities and initial electronic state. The initial geometry and velocities can be obtained in different ways. With SHARC, often sampling of a quantum-harmonic Wigner distribution is performed.

The sampling is carried out with the non-interactive Python script **wigner.py**. The theoretical background is summarized in section ??.

7.1.1 Usage

The general usage is

```
user@host> $SHARC/wigner.py [options] filename.molden
```

wigner.py takes exactly one command-line argument (the input file with the frequencies and normal modes), plus some options. Usually, the **-n** option is necessary, since the default is to only create 3 initial conditions.

The argument is the filename of the file containing the information about the vibrational frequencies and normal modes. The file is by default assumed to be in the MOLDEN formathttp://www.cmbi.ru.nl/molden/molden$_format.html$MOLDEN$format$. For usage with **wigner.py**, only the following blocks have to be present:

- [FREQ]
- [FR-COORD]
- [FR-NORM-COORD]

The script accepts a number of command-line options, specified in table **??**.

7.1.2 Normal mode types

The normal mode vectors contained in a MOLDEN file can follow different conventions, e.g., unscaled Cartesian displacements or different kinds of mass-weighted displacements. By default, **wigner.py** attempts to identify which convention is followed by the file (by performing different renormalizations and checking if the so-obtained matrix is orthogonal). In order to use this automatic detection, use **-f 0**, which is the default. Otherwise, there are four possible options: **-f 1** to assume normal modes in the GAUSSIAN convention (used by GAUSSIAN, TURBOMOLE, Q-CHEM, ADF, and ORCA); **-f 2** to assume Cartesian normal modes (used by MOLCAS and MOLPRO); **-f 3** to assume the COLUMBUS convention; or **-f 4** for mass-weighted, orthogonal normal modes.

7.1.3 Non-default masses

When the **-m** option is used, the script will ask the user to interactively modify the atom masses. For each atom (referred to by the atom index as in the MOLDEN file), a mass can be given (relative atomic weights). Note that the frequency calculation which produces the MOLDEN should be done with the same atomic masses.

7.1.4 Sampling at finite temperatures

When the **-t** option is used, the script assumes a finite, non-zero temperature. The sampling will then consist of two steps, where first randomly a vibrational state is picked from the Boltzmann distribution, and then the Wigner distribution of that state is employed. For more details, see Section **??**.

At high temperatures and for low-frequency modes it is possible that very large vibrational quantum numbers will be selected. Because of the occurrence of factorials in the Laguerre polynomials in the excited Wigner distributions, this leads to variable overflow for $\nu_{\text{vib}} > 150$. Hence, the highest vibrational quantum number considered is 150, and higher ones are set to 150. Since this can lead to an overrepresentation of $\nu_{\text{vib}} = 150$, with the **-T** option one can instead discard all samplings where $\nu_{\text{vib}} > 150$. No matter whether **-T** is used or not, keep in mind that usually such high vibrational states might invalidate the assumption of an harmonic oscillator, and other sampling methods (e.g., molecular dynamics) should be considered.

Table 7.1: Command-line options for script **wigner.py**.

Option	Description	Default
`-h`	Display help message and quit	—
`-n INTEGER`	Number of initial conditions to generate	3
`-m`	Modify atom masses (starts interactive dialog)	Most common isotopes
`-s FLOAT`	Scaling factor for the frequencies	1.0
`-t FLOAT`	Use Boltzmann-weighted distribution at the given temperature	0.0 K
`-T`	Discard very high vibrational states at high temperatures	Don't discard, but warn
`-L FLOAT`	Discard frequencies below the given one (in cm^{-1})	10.0
`-o FILENAME`	Output filename	**initconds**
`-x`	Creates an xyz file with the sampled geometries	**initconds.xyz**
`-l`	Instead of generating **initconds**, create input for **SHARC_LVC.py**	Create **initconds**
`-r INTEGER`	Seed for random number generator	16661
`-f F`	Type of normal modes read (0=detect automatically, 1–4=see below)	0
`--keep_trans_rot`	Do not remove translations and rotations from velocity vector	
`--use_eq_geom`	Sample only velocities, but keep equilibrium geometry	Sample normally
`--use_zero_veloc`	Sample only geometries, but set velocities to zero	Sample normally

7.1.5 Output

The script **wigner.py** generates a single output file, by default called **initconds**. All information about the initial conditions is stored in this file. Later steps in the preparation of the initial conditions add information about the excited states to this file. The file is formatted in a human-readable form.

The **initconds** file format is specified in section **??**.

When the **-x** option is given, additionally the script produces a file called **initconds.xyz**, which contains the sampled geometries in standard xyz format. This can be useful to inspect the distribution of geometry parameters (e.g., bond lengths) or to perform single point calculations at the sampled geometries.

When the **-l** option is given, the script only produces a file called **V0.txt**, which is a necessary input file for the LVC interface (see section **??**). If this option is activated, no **initconds** or **initconds.xyz** files are produced.

7.2 AMBER Trajectory Sampling: amber_to_initconds.py

The first step in preparing the dynamics calculation is to obtain a set of physically reasonable initial conditions. Each initial condition is a set of initial atomic coordinates, initial atomic velocities and initial electronic state. The initial geometry and velocities can be obtained in different ways. Besides sampling from a quantum mechanical Wigner distribution (with **wigner.py**), it is a widespread approach to sample geometries and velocities from a ground state molecular dynamics simulation.

Using **amber_to_initconds.py**, one can convert the results of an AMBER simulation to a SHARC **initconds** file.

7.2.1 Usage

In order to use **amber_to_initconds.py**, it is necessary to first carry out an AMBER simulation. You need to add the following options to the AMBER MD input file: (i) **ntxo=1** to tell AMBER to write ASCII restart files, (ii) **ntwr=-5000** to create a restart file every 5000 steps (other values are possible, but use the minus to not overwrite the restart files). This will create a set of AMBER restart files called, e.g., **md.rst_5000**, **md.rst_10000**, ...

Note that it is also necessary to reimage the AMBER restart files, because SHARC does not work with periodic boundary conditions, but the AMBER trajectories might use them. The reimaging can be performed with AMBER's tool **cpptraj**, using the following input:

```
parm <filename>.prmtop
trajin <filename>.rst7
autoimage
trajout <filename2>.rst7
run
```

This command has to be repeated for each restart file which needs to be reimaged. Note that **amber_to_initconds.py** only works with the **rst7** ASCII file format, not with the **rst** format (even if it is ASCII-formatted).

If you saved the restart files in AMBER's newer NetCDF format, **cpptraj** can also be used to convert them to ASCII **rst7** restart files. If you did not save restart files, **cpptraj** can even be used to generate restart files from the trajectory file (**.mdcrd** and **.mdvel**), but for this way it is necessary to save the velocities (**.mdvel** file).

With the restart files prepared, call **amber_to_initconds.py** like this:

```
user@host> $SHARC/amber_to_initconds.py [options] md.prmtop md.rst_0 md.rst_5000 md.rst_10000 ...
```

The possible options are shown in Table **??**.

7.2.2 Time Step

Note that the option **-t** (giving the time step used in AMBER in femtoseconds) is mandatory; if not given, an error message is produced. This is because AMBER uses a Leapfrog algorithm and thus stores in the restart file $R(t)$ and $v(t-\Delta t/2)$, whereas SHARC uses the velocity-Verlet algorithm and requires geometry and velocity at the same time, e.g., $R(t-\Delta t/2)$ and $v(t-\Delta t/2)$. To compensate this, **amber_to_initconds.py** computes $R(t-\Delta t/2)$ from $R(t)-v(t-\Delta t/2)\Delta t/2$. Hence, Δt of the AMBER trajectory needs to be known.

Table 7.2: Command-line options for script **amber_to_initconds.py**.

Option	Description	Default
`-h`	Display help message and quit	—
`-t FLOAT`	Time step (in femtoseconds) used in the AMBER simulation	—
`-o FILENAME`	Output filename	**initconds**
`-x`	Creates an xyz file with the sampled geometries	**initconds.xyz**
`-m`	Modify atom masses (starts interactive dialog)	As in **prmtop** file
`--keep_trans_rot`	Do not remove translations and rotations from velocity vector	
`--use_zero_veloc`	Sample only geometries, but set velocities to zero	Sample normally

7.2.3 Atom Types and Masses

By default, atom types and masses are read from the **prmtop** file (from flags **ATOMIC_NUMBER** and **MASS**). If the atomic number is not sensible (e.g., -1 for a transition metal) then **amber_to_initconds.py** prompts the user to define the element. The masses in the **prmtop** file can be overridden if the **-m** option is given; then the user can adjust the mass of each atom individually.

7.2.4 Output

amber_to_initconds.py produces the same output as **wigner.py** (section ??). By default, a file called **initconds** is generated for the converted initial conditions. It is important to note that the first restart file given (the second command line argument) is treated as the "equilibrium" geometry for the purpose of generating the **initconds** file. The second given restart file is then converted to the initial condition with index 1, and so on. Note that it is possible to give the same restart file multiple times as an argument (so that the same geometry can be used as "equilibrium" geometry and as proper initial condition.

7.3 SHARC Trajectory Sampling: sharctraj_to_initconds.py

The first step in preparing the dynamics calculation is to obtain a set of physically reasonable initial conditions. Each initial condition is a set of initial atomic coordinates, initial atomic velocities and initial electronic state. The initial geometry and velocities can be obtained in different ways. Besides sampling from a quantum mechanical Wigner distribution, it is often appropriate to sample geometries and velocities from a ground state molecular dynamics simulation.

Using **sharctraj_to_initconds.py**, one can convert the results of a SHARC simulation to a new SHARC **initconds** file.

7.3.1 Usage

In order to use **sharctraj_to_initconds.py**, it is necessary to first run a number of SHARC trajectories (the initial conditions for those need to be obtained with **wigner.py** or **amber_to_initconds.py**). The trajectories can be run with any number of states and in any state, and with any desirable options; only geometries and velocities are converted to the new **initconds** file.

With the trajectories prepared, call **sharctraj_to_initconds.py** like this:

```
user@host> $SHARC/sharctraj_to_initconds.py [options] Singlet_0 ...
```

Alternatively, with the **--give_TRAJ_paths** option, one can also do:

```
user@host> $SHARC/sharctraj_to_initconds.py --give_TRAJ_paths [options] TRAJ_00001 TRAJ_00002 ...
```

The possible options are shown in Table ??.

Table 7.3: Command-line options for script **sharctraj_to_initconds.py**.

Option	Description	Default
`-h`	Display help message and quit	—
`-r INTEGER`	Seed for the random number generator	16661
`-S INTEGER INTEGER`	Range of time steps from which a step is randomly chosen	last step
`-o FILENAME`	Output filename	**initconds**
`-x`	Creates an xyz file with the sampled geometries	**initconds.xyz**
`--keep_trans_rot`	Do not remove translations and rotations from velocity	
`--use_zero_veloc`	Sample only geometries, but set velocities to zero	Sample normally
`--debug`	Show timings	
`--give_TRAJ_paths`	Allows specifying individual trajectories	Specify parent directories

7.3.2 Random Picking of Time Step

For each directory specified as command line argument, **sharctraj_to_initconds.py** picks exactly one time step and extracts geometries and velocities of that time step. Note that a directory can be given several times as argument, so that multiple time steps can be selected.

The time steps are generally picked randomly (uniform probabilities) from an interval specified with the **-S** option. This option takes two integers, e.g., **-S 50 -50**, which can be positive or negative. The meaning of positive/negative/zero is the same as in PYTHON: positive numbers simply denote a time step (start counting with zero for the step zero of the trajectory); for negative numbers, start counting at the end, i.e., -1 is the last time step of the trajectory. In this way, it is possible to select the snapshot from the last *n* steps of all trajectories, even if they have different length. The example above, **-S 50 -50**, means picking a time step between the 50th and the 50th-last step. Note that if a trajectory is shorter than 100 steps, in this example it is skipped because there are no steps between the 50th and the 50th-last step.

7.3.3 Output

sharctraj_to_initconds.py produces the same output as **wigner.py** (section ??). By default, a file called **initconds** is generated for the converted initial conditions. It is important to note that the first directory given (the first command line argument) is treated as the "equilibrium" geometry for the purpose of generating the **initconds** file. The second given directory is then converted to the initial condition with index 1, and so on. Note that it is possible to give the same directory multiple times as an argument (so that the same geometry can be used as "equilibrium" geometry and as proper initial condition.

7.4 Setup of Initial Calculations: setup_init.py

The interactive script **setup_init.py** creates input for single point calculations at the initial geometries given in an **initconds** file. These calculations might be necessary for some schemes to select the initial electronic state of the trajectory, e.g., based on the excitation energies and oscillator strength of the transitions from ground state to the excited state, or based on overlaps with a reference wave function.

There are other choices of the initial state possible, which do not require single point calculations at all initial geometries. See the description of **excite.py** (section ??). In this case, **setup_init.py** can be used to set up only the calculation at the equilibrium geometry (see below at "Range of Initial Conditions").

7.4.1 Usage

The script is interactive, and can be started by simply typing

```
user@host> $SHARC/setup_init.py
```

Please be aware that the script will setup the calculations in the directory where it was started, so the user should **cd** to the desired directory before executing the script.

Please note that the script does not expand ~ or shell variables, except where noted otherwise.

7.4.2 Input

The script will prompt the user for the input. In the following, all input parameters are documented:

Initial Conditions File Enter the filename of the initial conditions file, which was generated beforehand with **wigner.py**. If the script finds a file called **initconds**, the user is asked whether to use this file, otherwise the user has to enter an appropriate filename. The script detects the number of initial conditions and number of atoms automatically from the initial conditions file.

Range of Initial Conditions The initial conditions in **initconds** are indexed, starting with the index 1. In order to prepare ab initio calculations for a subset of all initial conditions, enter a range of indices, e.g. a and b. This will prepare all initial conditions with indices in the interval $[a, b]$. In any case, the script will additionally prepare a calculation for the equilibrium geometry (except if a finished calculation for the equilibrium geometry was found).

If the interval $[0, 0]$ is given, the script will only setup the calculation at the equilibrium geometry.

Number of states Here the user can specify the number of excited states to be calculated. Note that the ground state has to be counted as well, e.g., if 4 singlet states are specified, the calculation will involve the S_0, S_1, S_2 and S_3. Also states of higher multiplicity can be given, e.g. triplet or quintet states. For even-electron molecules, including odd-electron states (e.g. doublets) is only useful if transition properties for ionization can be computed (e.g. Dyson norms with some of the interfaces). These transition properties can be used to calculate ionization spectra or to obtain initial conditions for dynamics after ionization.

Interface In this point, choose any of the displayed interfaces to carry out the ab initio calculations. Enter the corresponding number.

Spin-orbit calculation Usually, it is sufficient to calculate the spin-free excitation energies and oscillator strengths in order to decide for the initial state. However, using this option, the effects of spin-orbit coupling on the excitation energies and oscillator strengths can be included. Note that the script will never calculate spin-orbit couplings if only singlet states are included in the calculation, or if the chosen interface does not support calculation of spin-orbit couplings.

Reference overlaps The calculations can be setup in such a way that the wave function overlaps between states at the equilibrium geometry and the displaced geometries is computed. This allows correlating the states at the displaced geometries with the reference states, such that one can know the state characters of all states without inspection. This is useful for a crude "diabatization" of the states, e.g., if one wants to start all trajectories in the $n\pi^*$ state of the molecule although this state can be S_1, S_2, or S_3 (use **excite.py** to setup initial conditions in such a way, see section **??**).

When activating this option, keep in mind that the calculation in **ICOND_00000** must be successfully finished before any of the other **ICOND_XXXXX** calculations can be started.

TheoDORE analysis If the chosen interface supports wave function analysis with TheoDORE, then this option can be activated here. **setup_init.py** will then include the relevant keywords in the computations, and the results of the TheoDORE analysis will be written to the **QM.out** files.

The remaining settings for TheoDORE (fragment and descriptor input) will be asked later in **setup_init.py**.

7.4.3 Interface-specific input

The following input in **setup_init.py** depends on the chosen interface. However, some input is similar between several interfaces and is discussed (out of input order) in section **??**. Note that most of the questions asked in the input dialogue have some counterpart in the resource file of the chosen interface, so you might find additional information in chapter **??**.

Input which is similar between interfaces

Path to quantum chemistry programs and licenses Here the user is prompted to provide the path to the relevant executables or installation directories.

Note that the setup script will not expand the user (~) and shell variables (since the calculations might be running on a different machine than the one used for setup, so the meaning of shell variables could be different). ~ and shell variables will only be expanded by the interfaces during the actual calculation.

For some interfaces, also the path to a license file might need to be specified.

Scratch directory The script takes the string without any checking. Each individual initial condition uses a subdirectory of the given path, so that there are no clashes between the initial conditions. ~ and shell variables will only be expanded by the interface during the actual calculation.

Note that you can use, e.g., **./temp/** as scratch directory, which is a subdirectory of the working directory of the interface. Using **./** as scratch directory is not recommended, since the interface will delete the scratch directory.

Template file For almost all interfaces, **setup_init.py** will ask for a template file containing the quantum chemistry specifics of the calculations. The format of the template file depends on the interface; formats are described in the interface chapter ??.

For most interfaces, **setup_init.py** will perform some basic checks, but ultimately the template file will be checked by the interface once the calculation is submitted.

Initial orbital guess For some interfaces, **setup_init.py** will ask for files containing initial orbital guesses. providing initial is especially important for multi-reference calculations, because otherwise it is very likely that the calculations converge to an undesirable active space.

Resource usage For most interfaces, **setup_init.py** will ask for the amount of memory and the number of CPU cores to use. For some interfaces, additionally a delay between the start of parallel calculation can be provided (this might be helpful if many calculations starting at the same time is problematic for I/O). For the MOLCAS, ADF, **Gaussian**, and ORCA interfaces, furthermore the **schedule_scaling** is queried for (see section ?? for details).

Wavefunction overlap program The scripts asks for the path to the wavefunction overlap program. In a complete SHARC installation, the overlap program should be located at **$/SHARC/wfoverlap.x**. Depending on the interface, the script might also ask for the threshold to truncate the wave function files, or for the memory for **wfoverlap.x** (if the memory of the quantum chemistry program cannot be set). For some interfaces, it is also possible to control the number of frozen core orbitals for the overlap calculation (and the number of inactive orbitals for Dyson orbital calculations).

THEODORE setup If THEODORE is enabled, **setup_init.py** will query for the path to the THEODORE installation directory. Furthermore, the user needs to enter a list of the descriptors to be calculated by THEODORE, as well as the atom indices for each fragment for the charge transfer number computation.

QM/MM setup For some interfaces, **setup_init.py** will ask for QM/MM-related files if the template file specifies a QM/MM calculation. Currently, the QM/MM setup consists simply of providing the relevant interface-specific files to **setup_init.py**, which it simply copies accordingly.

Input for MOLPRO

Path to MOLPRO Here the user is prompted to provide the path to the MOLPRO executable. For more details, see section ??.

Scratch directory See section ??.

Template file Enter the filename for the MOLPRO template input. This file should contain at least the basis set, active orbitals and electrons, number of electrons and number of states for state-averaging. For details, see the section about the SHARC-MOLPRO interface (??). The setup script will check whether the template file contains the necessary entries.

Initial wave function file You can provide an initial wave function (with an appropriate MO guess) to MOLPRO. This usually provides a drastic speedup for the CASSCF calculations and helps in ensuring that all calculations are based on the same active space. In simple cases it might be acceptable to not provide an initial wave function.

Resource usage The script asks for the memory and number of CPU cores to be used. Note that both settings apply to MOLPRO and to `wfoverlap.x`, if the latter is required (but note that the two programs are never run simultaneously). If more than one CPU core is used, the interface will parallelize different MOLPRO runs (but will not run MOLPRO in parallel mode); `wfoverlap.x` will be run in SMP-parallel mode.

Wavefunction overlap program See section ??. Note that for the MOLPRO interface, no truncation threshold can be given, since `wfoverlap.x` is very efficient for CASSCF wavefunctions.

Input for MOLCAS

Path to MOLCAS Here the user is prompted to provide the path to the MOLCAS directory. For more details, see section ??.

Scratch directory See section ??.

Template file Enter the filename for the MOLCAS template input. This file should contain at least the basis set, active orbitals and electrons, number of inactive orbitals, and number of states for state-averaging. For details, see the section about the SHARC-MOLCAS interface (??). The setup script will check whether the template file contains the necessary entries.

QM/MM If the keyword `qmmm` is contained in the template file, the user will be queried for the path to TINKER's `bin/` directory. Note that only TINKER installations with the necessary modifications for MOLCAS can be used.

The script will also ask for the key file (containing the path to the force field file and the QM/MM partition) and the connection table file. See section ?? for details.

Initial wave function file You can provide initial MO coefficients to MOLCAS. This usually provides a drastic speedup for the CASSCF calculations and helps in ensuring that all calculations are based on the same active space. In simple cases it might be acceptable to not provide an initial wave function. For MOLCAS, you have to specify one file for each multiplicity (but you can use the same file as guess for several multiplicities). Initial orbital files can either be in `JobIph` or `RasOrb` format.

Resource usage The script asks for the memory and number of CPU cores to be used. Note that both settings apply to MOLCAS and to `wfoverlap.x`, if the latter is required (but note that the two programs are never run simultaneously). If more than one CPU core is used, the interface will parallelize different MOLCAS runs (but will not run MOLCAS in parallel mode); `wfoverlap.x` will be run in SMP-parallel mode.

Wavefunction overlap program See section ??. The `wfoverlap.x` program is only required if Dyson norms need to be calculated. Note that for the MOLCAS interface, no truncation threshold can be given, since `wfoverlap.x` is very efficient for CASSCF wavefunctions.

Input for COLUMBUS

Path to COLUMBUS Here the user is prompted to provide the path to the COLUMBUS directory. For more details, see section ??.

Scratch directory See section ??.

Template directory Enter the path to a directory containing subdirectories with COLUMBUS input files necessary for the calculations. The setup script will expand ~ and shell variables and will pass the absolute path to the calculations.

The script will auto-detect (based on the **cidrtin** files) which subdirectory contains the input for each multiplicity. The user has to check in the following dialog whether the association of multiplicities with job directories is correct. Additionally, the association of MO coefficient files to the jobs has to be checked. See the section on the SHARC-COLUMBUS interface (??) for further details.

Note that the setup script does not check any content of the input files beyond the multiplicity. Note that **transmomin** and **control.run** do not need to be present (and that their content has no effect on the calculation), since these files are written on-the-fly by the interface.

The template directory can either be linked or copied to the initial condition calculations.

Initial orbital coefficient file You can provide initial MO coefficients for the COLUMBUS calculation. This usually provides a drastic speedup for the CASSCF calculations and helps to ensure that all calculations are based on the same active space. In simple cases it might be acceptable to not provide an initial wave function.

Memory For COLUMBUS, the available memory must be given here. The COLUMBUS interface does not allow for parallelization currently.

Wavefunction overlap program See section ??. For CASSCF calculations with COLUMBUS, use a truncation threshold of 1.0, for MRCIS/MRCISD calculations, 0.97-0.99 should provide sufficient performance and accuracy.

Input for analytical potentials

Template file Enter the filename for the template input specifying the analytical potentials. For details, see the section about the SHARC-Analytical interface (??). The setup script will check whether the template file is valid.

Input for ADF

Path to ADF Here the user is first asked if setup should be made from an **adfrc.sh** file. If yes, only the path to this file is needed.

Otherwise, the path to the ADF installation directory and the path to the license file need to be provided. For more details, see section ??.

Scratch directory See section ??.

Template file Enter the filename for the ADF template input. This file should contain at least the basis set, functional, and charge keywords. For details, see the section about the SHARC-ADF interface (??). The setup script will check whether the template file contains the necessary entries.

QM/MM The script will ask for the two QM/MM-specific input files for the interface, which are called **ADF.qmmm.ff** and **ADF.qmmm.table**. See section ?? for details on these files.

Initial orbital coefficient file You can provide initial MO coefficients for the ADF calculation. This can provide a small speedup for the calculations and helps to ensure that all calculations are based on the same orbitals. In many cases, no initial orbitals are required.

Resource usage For ADF, the amount of memory to be used cannot be controlled. However, the number of CPU cores needs to be given. If more than one core is used, also the parallel scaling needs to be specified (see section ??).

Wavefunction overlap program See section ??. **setup_init.py** will ask for the memory usage of **wfoverlap.x**. The truncation threshold for TD-DFT can usually be chosen as 0.99–1.00 (depending on the system and functional, but it is recommended that it is high enough that each state is described by at least 100 determinants in the generated files). See section ?? for details.

THEODORE setup See section ??.

Input for RICC2

Path to TURBOMOLE The path to the TURBOMOLE installation directory needs to be provided. If spin-orbit couplings are requested, also the path to the ORCA installation directory is needed. For more details, see section ??.

Scratch directory See section ??.

Template file Enter the filename for the RICC2 template input. This file should contain at least the basis set and charge keywords. For details, see the section about the SHARC-RICC2 interface (??). The setup script will check whether the template file contains the necessary entries.

Initial orbital coefficient file You can provide initial MO coefficients for the TURBOMOLE calculation. This can provide a small speedup for the calculations and helps to ensure that all calculations are based on the same orbitals. In many cases, no initial orbitals are required.

Resource usage The amount of memory and the number of CPU cores are required in this section. Note that the SHARC-RICC2 interface always runs only one TURBOMOLE instance at the same time (if the number of CPU cores is larger than one, it will use the SMP/OMP binaries of TURBOMOLE).

Wavefunction overlap program See section ??. **setup_init.py** will ask for the memory usage of **wfoverlap.x**. The truncation threshold can usually be chosen as 0.99–1.00 (depending on the system, but it is recommended that it is high enough that each state is described by at least 100 determinants in the generated files). See section ?? for details.

THEODORE setup See section ??.

QM/MM The script will ask for the two QM/MM-specific input files for the interface, which are called **RICC2.qmmm.ff** and **RICC2.qmmm.table**. See section ?? for details on these files.

Input for LVC interface

Template file Enter the filename for the template input specifying the LVC parameters. For details, see the section about the SHARC-LVC interface (??). The setup script will not check whether the template file is valid.

Input for GAUSSIAN interface

Path to GAUSSIAN The path to the GAUSSIAN installation directory needs to be provided. Note that this needs to be the path to the directory containing the GAUSSIAN executables, e.g., **g09**/**g16**, **l9999.exe**, etc. The interface will automatically detect which GAUSSIAN version is used. For more details, see section ??.

Scratch directory See section ??.

Template file Enter the filename for the GAUSSIAN template input. This file should contain at least the basis set, functional, and charge keywords. For details, see the section about the SHARC-GAUSSIAN interface (??). The setup script will check whether the template file contains the necessary entries.

Initial orbital coefficient file You can provide initial MO coefficients for the GAUSSIAN calculation. This can provide a small speedup for the calculations and helps to ensure that all calculations are based on the same orbitals. In many cases, no initial orbitals are required.

Resource usage The amount of memory and the number of CPU cores are required in this section. If more than one core is used, also the parallel scaling needs to be specified (see section ??).

Wavefunction overlap program See section ??. **`setup_init.py`** will ask for the memory usage of **`wfoverlap.x`**. The truncation threshold for TD-DFT can usually be chosen as 0.99–1.00 (depending on the system and functional, but it is recommended that it is high enough that each state is described by at least 100 determinants in the generated files). See section ?? for details.

THEODORE setup See section ??.

Input for ORCA interface

Path to ORCA The path to the ORCA installation directory needs to be provided. Note that this needs to be the path to the directory containing the ORCA executables, e.g.,**`orca`** or **`orca_fragovl`**. For more details, see section ??.

Scratch directory See section ??.

Template file Enter the filename for the ORCA template input. This file should contain at least the basis set, functional, and charge keywords. For details, see the section about the SHARC-ORCA interface (??). The setup script will check whether the template file contains the necessary entries.

Initial orbital coefficient file You can provide initial MO coefficients for the ORCA calculation. This can provide a small speedup for the calculations and helps to ensure that all calculations are based on the same orbitals. In many cases, no initial orbitals are required.

Resource usage The amount of memory and the number of CPU cores are required in this section. If more than one core is used, also the parallel scaling needs to be specified (see section ??).

Wavefunction overlap program See section ??. **`setup_init.py`** will ask for the memory usage of **`wfoverlap.x`**. The truncation threshold for TD-DFT can usually be chosen as 0.99–1.00 (depending on the system and functional, but it is recommended that it is high enough that each state is described by at least 100 determinants in the generated files). See section ?? for details.

THEODORE setup See section ??.

QM/MM The script will ask for the two QM/MM-specific input files for the interface, which are called **`ORCA.qmmm.ff`** and **`ORCA.qmmm.table`**. See section ?? for details on these files.

Input for BAGEL interface

Path to BAGEL The path to the BAGEL installation directory needs to be provided. Note that this needs to be the path to the main directory of BAGEL, with subdirectories like **`bin/`** or **`lib/`**. For more details, see section ??.

Scratch directory See section ??.

Template file Enter the filename for the BAGEL template input. This file should contain at least the basis set, active space, and charge keywords. For details, see the section about the SHARC-BAGEL interface (??). The setup script will check whether the template file contains the necessary entries.

Initial orbital coefficient file You can provide initial MO coefficients for the BAGEL calculation. This usually provides a drastic speedup for the CASSCF calculations and helps to ensure that all calculations are based on the same active space. In simple cases it might be acceptable to not provide an initial wave function.

Resource usage The amount of memory and the number of CPU cores are required in this section. If more than one core is used, also the parallel mode needs to be specified (MPI or OpenMP).

Wavefunction overlap program See section ??. **setup_init.py** will ask for the memory usage of **wfoverlap.x**. The truncation threshold for TD-DFT can usually be chosen as 0.99–1.00 (depending on the system and functional, but it is recommended that it is high enough that each state is described by at least 100 determinants in the generated files). See section ?? for details.

7.4.4 Input for Run Scripts

Run script mode The script **setup_init.py** generates a run script (Bash) for each initial condition calculation. Due to the large variety of cluster architectures, these run scripts might not work in every case. It is the user's responsibility to adapt the generated run scripts to his needs.

setup_init.py can generate run scripts for two different schemes how to execute the calculations. With the first scheme, the ab initio calculations are performed in the directory where they were setup (subdirectories of the directory where **setup_init.py** was started). Note that the interfaces will still use their scratch directories to perform the actual quantum chemistry calculations. Currently, this is the default option.

With the second option, the run scripts will transfer the input files for each ab initio calculation to a temporary directory, where the interface is started. After the interface finishes all calculations, the results files are transferred back to the primary directory and the temporary directory is deleted. Note that **setup_init.py** in any case creates the directory structure in the directory where it was started. The name of the temporary directory can contain shell variables, which will be expanded when the script is running (on the compute host).

Submission script The setup script can also create a Bash script for the submission of all ab initio calculations to a queueing system. The user has to provide a submission command for that, including any options which might be necessary. This submission script might not work with all queueing systems.

Project name The user can enter a project name. This is used currently only for the job names of submitted jobs (**-N** option for queueing system).

7.4.5 Output

setup_init.py will create for each initial condition in the given range a directory whose names follow the format **ICOND_%05i/**, where **%05i** is the index of the initial condition padded with zeroes to 5 digits. Additionally, the directory **ICOND_00000/** is created for the calculation of the excitation energies at the equilibrium geometry.

To each directory, the following files will be added:

- **QM.in**: Main input file for the interface, contains the geometry and the control keywords (to specify which quantities need to be calculated).
- **run.sh**: Run script, which can be started interactively in order to perform the ab initio calculation in this directory. Can also be adapted to a batch script for submission to a queue
- Interface-specific files: Usually a template file, a resource file, and an initial wave function.

The calculations in each directory can be simply executed by starting **run.sh** in each directory. In order to perform this task consecutively on a single machine, the script **all_run.sh** can be executed. The file **DONE** contains the progress of this calculation. Alternatively, each run script can be sent to a queueing system (you might need to adapt this script to you cluster system). Note that if reference overlaps were requested, the calculation in **ICOND_00000/** must be finished before starting any of the other calculations.

In figure ??, the directory tree structure setup by **setup_init.py** is given.

After all calculations are finished, **excite.py** can be used to collect the results.

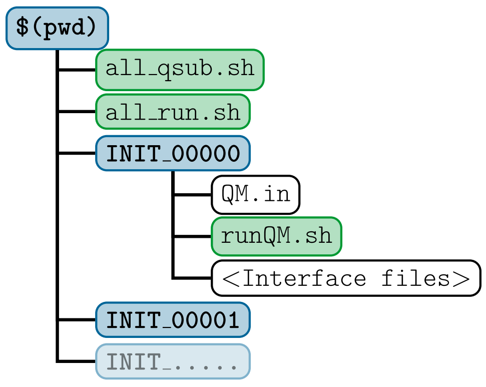

Figure 7.1: Directory structure created by **setup_init.py**. Directories are in blue, executable scripts in green and regular files in black and white. Interface files usually include initial MO coefficients, template files and interface input files.

7.5 Excitation Selection: excite.py

excite.py has two tasks: adding excited-state information to the **initconds** file, and deciding which excited state for which initial condition is a valid initial state for the dynamics.

7.5.1 Usage

The script is interactive, and can be started by simply typing

```
user@host> $SHARC/excite.py
```

7.5.2 Input

Initial condition file Enter the path to the initial conditions file, to which **excite.py** will add excited-state information. This file can already contain excited-state information (in this case this information can be reused).

Generate excited state list There are three possibilities to add excited-state information to the **initconds** file:

1. generate a list of dummy excited states,
2. read excited-state information from the output of the initial ab initio calculations (prepare the calculations with **setup_init.py**),
3. keep the existing excited-state information in the **initconds** file.

The first option is mainly used if no initial ab initio calculations need to be performed (e.g., the initial state is known).

In order to use the second option, one should first setup initial excited-state calculations using **setup_init.py** (see ??) and run the calculations. **excite.py** can then read the output of the initial calculations and calculate excitation energies and oscillator strengths.

The third option can be used to reuse the information in the **initconds** file, e.g., to apply a different selection scheme to the states or to just read the number of states.

Path to ab initio results If **excite.py** will read the excited-state information from the ab initio calculation results, here the user has to provide the path to the directory containing the **ICOND_%05i** subdirectories.

Number of states If a dummy list of states will be generated, the user has to provide the number of states per multiplicity. Note that a singlet ground state has to be counted as well, e.g. if 4 singlet states are specified, the calculation will involve the S_0, S_1, S_2 and S_3. Also states of higher multiplicity can be given, e.g. doublet or triplet states (e.g., **2 2 1** for two singlets, two doublets and one triplet).

If the ab initio results are read the number of states will be automatically determined from the results.

Excited-state representation When generating new lists of excited states (either dummy states or from ab initio results), the user has to specify the representation of the excited states (either MCH or diagonal representation). The MCH representation is spin-free, meaning that transition dipole moments are only allowed between states of the same multiplicity. For molecules without heavy atoms, this option is sufficient. For heavier atoms, the diagonal representation can be used, which includes the effects of spin-orbit coupling on the excitation energies and oscillator strengths. Note, however, that excited-state selection with delta pulse excitation (option 3 under "Initial state selection") should be carried out in the MCH representation if the ground state is not significantly spin-orbit-mixed.

When reading ab initio results, **excite.py** will diagonalize the Hamiltonian and transform the transition dipole matrices for each initial condition to obtain the diagonal representation.

When a dummy state list is generated, the representation will only be written to **initconds.excited** (but has no actual numeric effect for **excite.py**). Note that the representation which is declared in the **initconds.excited** file influences how SHARC determines the initial coefficients (see the paragraph on initial coefficients in ??).

Note that the representation cannot be changed if existing excited-state information is kept.

Hint: If the **ICOND_%05i** directories need to be deleted (e.g., due to disk space restrictions), making one read-out with **excite.py** for each representation and saving the results to two different files will preserve most necessary information.

Ionization probabilities If **excite.py** detects that the ab initio results contain ionization probabilities, then those can be used instead of the transition dipole moments. Note that in this case the transition dipole moments are not written to the **initconds.excited** file.

Reference energy **excite.py** can read the reference energy (ground state equilibrium energy) directly from the ab initio results. If the ab initio data is read anyways, **excite.py** already knows the relevant path. If a dummy list of states is generated, the user can provide just the path to the **QM.out** file of the ab initio calculation for the equilibrium geometry. Otherwise, **excite.py** will prompt the user to enter a reference energy manually (in hartree).

Initial state selection Every excited state of each initial condition has a flag specifying it either as a valid initial state or not. **excite.py** has four modes how to flag the excited states:

1. Unselect all excited states,
2. User provides a list of initial states,
3. States are selected stochastically based on excitation energies and oscillator strengths,
4. Keep all existing flags.

The first option can be used if **excite.py** is only used to read the ab initio results for the generation of an absorption spectrum (using **spectrum.py**).

The second option can be used to directly specify a list of initial states, if the initial state is known (e.g., starting in the ground state and exciting with an explicit laser field). In this case, the given states of *all* initial conditions are flagged as initial states. This option is also useful if reference overlaps were computed (see **??**.

The third option is only available if excited-state information exists (i.e., if no dummy list is generated). For details on the stochastic selection procedure, see section **??**.

The fourth option can only be used if the existing state information is kept. In this case **excite.py** does nothing except counting the number of flagged initial states.

Excitation window This option allows to exclude excited states from the selection procedure if they are outside a given energy window. This option is only available if excited state information exists, but not if a dummy list of states is generated (because the dummy states have no defined excitation energy).

For the stochastic selection procedure, states outside the excitation window do not count for the determination of $p_{\max}$ (see equation (??)). This allows to excite, e.g., to a dark $n\pi^*$ state despite the presence of a much brighter $\pi\pi^*$ state.

For the keep-flags option, this option can be used to count the number of excited states in the energy window.

Considered states Here the user can specify the list of desired initial states. If reference overlaps are present in the excitation calculations, then the user can choose to specify the initial state in terms of diabatized states (as defined by the overlap with the reference, where the diabatized states are identical to the computed states). See section **??** for how the diabatization is carried out.

For the stochastic selection procedure, the user can instead exclude certain states from the procedure. Excluded states do not count for the determination of $p_{\max}$ (see equation (**??**)).

If the number of states per multiplicity is known, **excite.py** will print a table giving for each state index the multiplicity, quantum number and M_s value.

Random number generator seed The random number generator in **excite.py** is used in the stochastic selection procedure. Instead of typing an integer, typing "**!**" will initialize the RNG from the system time. Note that this will not be reproducible, i.e. repeating the **excite.py** run with "**!**" as random seed will give a different selection in each run.

7.5.3 Matrix diagonalization

When using the diagonal representation, **excite.py** needs to diagonalize and multiply matrices. By default, the Python package NUMPY is used, if available. If the script does not find a NUMPY installation, it will use a small Fortran code which comes with the SHARC suite. In order for this to work, you need to set the environment variable **$SHARC** to the **bin/** directory within your SHARC installation. See section **??** for more details.

7.5.4 Output

excite.py writes all output to a file **<BASE>.excited**, where **<BASE>** is the name of the initial conditions file used as input. The output file is also an initial conditions file, but contains additional information regarding the excited states, the reference energy and the representation of the excited states. An initial conditions file with excited-state information is needed for the final preparatory step: setting up the dynamics with **setup_traj.py**. Additionally, **spectrum.py** can calculate absorption spectra from excited-state initial condition files.

7.5.5 Specification of the initconds.excited file format

The initial conditions files **initconds** and **initconds.excited** contain lists of initial conditions, which are needed for the setup of trajectories. An initial condition is a set of initial coordinates of all atoms and corresponding initial velocities of each atom, and optionally a list of excited state informations. In the following, the format of this file is specified.

The file contains of a header, followed by the body of the file containing a list of the initial conditions.

File header An examplary header looks like:

```
SHARC Initial conditions file, version 0.2   <Excited>
Ninit     100
Natom     2
Repr      MCH
Eref         -0.50
Eharm           0.04
States    2 0 1

Equilibrium
 H   1.0  0.0  0.0  0.0   1.00782503   0.0  0.0  0.0
 H   1.0  1.5  0.0  0.0   1.00782503   0.0  0.0  0.0
```

The first line must read **SHARC Initial conditions file, version <VERSION>**, with the correct version string followed. The string **Excited** is optional, and marks an initial conditions file as being an output file of **excite.py** (**setup_traj.py** will only accept files marked like this). The following lines contain:

1. the number of initial conditions,
2. the number of atoms,
3. the electronic state representation (a string which is **None**, **MCH** or **diag**),
4. the reference energy (hartree),
5. the harmonic energy (zero point energy in the harmonic approximation, hartree),
6. optionally the number of states per multiplicity.

After the header, first the equilibrium geometry is expected. It is demarked with the keyword **Equilibrium**, followed by n_{atom} lines, each specifying one atom. Unlike the actual initial conditions, the equilibrium geometry does not have a list of excited states or defined energies.

File body The file body contains a list of initial conditions. Each initial condition is specified by a block starting with a line containing the string **Index** and the number of the initial condition. In the file, the initial conditions are expected to appear in order.

A block specifying an initial condition looks like:

```
Index     1
Atoms
 H   1.0  -0.02  0.0  0.0   1.00782503  -0.001  0.0  0.0
 H   1.0   1.52  0.0  0.0   1.00782503   0.001  0.0  0.0
States
001    -0.49    -0.49  -0.16   0.0  -0.03   0.0   0.05   0.0   0.0   0.00 False
002    -0.25    -0.49   0.02   0.0   0.43   0.0  -1.77   0.0   6.5   0.53 True
003    -0.40    -0.49   0.00   0.0   0.00   0.0   0.00   0.0   2.5   0.00 False
004    -0.40    -0.49   0.00   0.0   0.00   0.0   0.00   0.0   2.5   0.00 False
005    -0.40    -0.49   0.00   0.0   0.00   0.0   0.00   0.0   2.5   0.00 False
Ekin        0.004 a.u.
Epot_harm   0.026 a.u.
Epot        0.013 a.u.
Etot_harm   0.030 a.u.
Etot        0.018 a.u.
```

The formal structure of such a block is as follows. After the line containing the keyword **Index** and the index number, the keyword **Atoms** indicates the start of the list of atoms. Each atom is specified on one line:

1. symbol,
2. nuclear charge,
3. x, y, z coordinate in Bohrs,
4. atomic mass,
5. x, y and z component of nuclear velocity in atomic units.

After the atom list, the keyword **States** indicates the list of electronic states. This list consists of one line per electronic state, but can be empty, if no information of the electronic states is available. Each line consists of:

1. state number (starting with 1),
2. state energy in Hartree,
3. reference energy in Hartree (usually the energy of the lowest state),
4. six numbers defining the transition dipole moment to the reference state (usually the lowest state),
5. the excitation energy in eV,
6. the oscillator strength,
7. a string which is either **True** or **False**, specifying whether the electronic state was selected by **excite.py** as initial electronic state.

The transition dipole moments are specified by six floating point numbers, which are real part of the x component, imaginary part of the x component, then the real and imaginary parts for the y and finally the z component (the transition dipole moments can be complex in the diagonal representation).

The electronic state list is terminated with the keyword **Ekin**, which at the same time gives the kinetic energy of all atoms. The remaining entries give the potential energy in the harmonic approximation and the actual potential energy, as well as the total energy.

7.6 Setup of Trajectories: `setup_traj.py`

This interactive script prepares the input for the excited-state dynamics simulations with SHARC. It works similarly to **setup_init.py**, reading an initial conditions file, prompting the user for a number of input parameters, and finally prepares one directory per trajectory. However, the **setup_traj.py** input section is noticeably longer, because most options for the SHARC dynamics are covered.

7.6.1 Input

Initial conditions file Please be aware that **`setup_traj.py`** needs an initial conditions file generated by **`excite.py`** (files generated by **`wigner.py`**, **`amber_to_initconds.py`**, or **`sharctraj_to_initconds.py`** are not allowed). The script reads the number of initial states, the representation, and the reference energy automatically from the file.

Number of states This is the total number of states per multiplicity included in the dynamics calculation. Affects the keyword **`nstates`** in the SHARC input file.

Only advanced users should use here a different number of states than given to **`setup_init.py`**. In this case, the excited-state information in the initial conditions file might be inconsistent. For example, if 10 singlets and 10 triplets were included in the initial calculations, but only 5 singlets and 5 triplets in the dynamics, then the sixth entry in the initial conditions file corresponds to S_5, while **`setup_traj.py`** assumes the sixth entry to correspond to T_1.

Active states States can be frozen for the dynamics calculation here. See section **??** for a general description of state freezing in SHARC. Only the highest states in each multiplicity can be frozen, it is not possible to, e.g., freeze the ground state in simulations where ground state relaxation is negligible. Affects the keyword **`actstates`**.

Contents of the initial conditions file Optionally, a map of the contents of the initial conditions file can be displayed during the execution of **`setup_traj.py`**, showing for each state which initial conditions were selected (and which initial conditions do not have the necessary excited-state information). For each state, a table is given, where each symbol represents one initial condition. A dot "`.`" represents an initial condition where information about the current excited state is available, but which is not selected for dynamics. A hash mark "**`#`**" represents an initial condition which is selected for dynamics. A question mark "**`?`**" represents initial conditions for which no information about the excited state is available (e.g. if the initial excited-state calculation failed). The tutorial shows an example of this output.

The content of the initial conditions file is also summarized in a table giving the number of initial conditions selected per state.

Initial states for dynamics setup The user has to input all states from which trajectories should be launched. The numbers must be entered according to the above table giving the number of selected initial conditions per state. It is not allowed to specify inactive states as initial states. The script will give the number of trajectories which can be setup with the specified set of states. If no trajectories can be setup, the user has to specify another set of initial states. The initial state will be written to the SHARC input, specified in the same representation as given in the initial conditions file. The initial coefficients will be determined automatically by SHARC, according to the description in section **??**.

Starting index for dynamics setup Specifies the first initial condition within the initial condition file to be included in the setup. This is useful, for example, if the user might setup 50 trajectories starting with index 1. **`setup_traj.py`** reports afterwards the last initial condition to be used for setup, e.g. index 90. Later, the user can setup additional trajectories, starting with index 91.

Random number generator seed The random number generator in **`setup_traj.py`** is used to randomly generate RNG seeds for the SHARC input. Instead of typing an integer, typing "**`!`**" will initialize the RNG from the system time. Note that this will not be reproducible, i.e. repeating the **`setup_traj.py`** run (with the same input) with "**`!`**" as random seed will give for the same trajectories different RNG seeds. Affects the keyword **`RNGseed`**.

Interface In this point, choose any of the displayed interfaces to carry out the ab initio calculations. Enter the corresponding number. The choice of the interface influences some dynamics options which can be set in the next section of the **`setup_traj.py`** input.

Simulation Time This is the maximum time that SHARC will run the dynamics simulation. If trajectories need to be run for longer time, it is recommended to first let the simulation finish. Afterwards, increase the simulation time in the corresponding SHARC input file (keyword **`tmax`**) and add the restart keyword (also make sure that the **`norestart`** keyword is not present). Then the simulation can be restarted by running again the **`run.sh`** script. Sets the keyword **`tmax`** in the SHARC input files.

Simulation Timestep This gives the time step for the dynamics. The on-the-fly ab initio calculations are performed with this time step, as is the propagation of the nuclear coordinates. A shorter time step gives more accurate results, especially if light atoms (hydrogen) are subjected to high kinetic energies or steep gradients. Of course a shorter time step is computationally more expensive. A good compromise in many situations is 0.5 fs. Sets the keyword **stepsize** in the SHARC input files.

Number of substeps This gives the number of substeps for the interpolation of the Hamiltonian for the propagation of the electronic wave function. Usually, 25 substeps are sufficient. In cases where the diagonal elements of the Hamiltonian are very large (very large excitation energies or a badly chosen reference energy) more substeps are necessary. Sets the keyword **nsubsteps** in the SHARC input files.

Prematurely terminate trajectories Usually, trajectories which relaxed to the ground state do not recross to an excited state, but vibrate indefinitely in the ground state. If the user is not interested in these vibrations, such trajectories can be terminated prematurely in order to save computational resources. A threshold of 10–20 fs is usually a good choice to safely detect ground state relaxation. Sets the keyword **killafter** in the SHARC input files.

Representation for the dynamics Either the diagonal representation can be chosen (by typing "**yes**") to perform dynamics with the SHARC methodology, or the dynamics can be performed on the MCH states (spin-diabatic dynamics [**?**], FISH [**?**]). Sets the keyword **surf** in the SHARC input files.

Spin-orbit couplings If more than just singlet states are requested, the script asks whether spin-orbit couplings should be computed. If the chosen interface cannot provide spin-orbit couplings, this question is automatically answered.

Non-adiabatic couplings Electronic propagation can be performed with temporal derivatives, nonadiabatic coupling vectors or overlap matrices (Local diabatization). Enter the corresponding number. Note that depending on the chosen interface, some options might not be available, as displayed by **setup_traj.py**. Also note that currently, no interface can provide temporal derivatives (because their computation involves calculating the overlap matrix and then local diabatization can be done instead). Sets the keyword **coupling** in the SHARC input files.

If nonadiabatic coupling vectors are chosen, the user is asked whether overlap matrices should be computed anyways to provide wave function phase information. As the overlap calculations are usually fast compared to other steps, this is recommended.

Gradient transformation The nonadiabatic coupling vectors can be used to correctly transform the gradients to the diagonal representation. If nonadiabatic coupling vectors are used anyways, this option is strongly recommended, since it gives more accurate gradients for no additional cost. Sets the keyword **gradcorrect** in the SHARC input files. If the dynamics uses the MCH representation, this question is not asked.

Surface hop treatment This option determines how the total energy is conserved after a surface hop and whether frustrated hops lead to reflection. Sets the keywords **ekincorrect** and **reflect_frustrated** in the SHARC input files.

Decoherence correction For most applications, a decoherence correction should be enabled. This controls the **decoherence_scheme** (and **decoherence_param** keywords in the SHARC input files.

Note that **setup_traj.py** does not allow to modify the α parameter for the energy-based decoherence (keyword **decoherence_param**). In order to change **decoherence_param**, the user has to manually edit the SHARC input files.

Surface hopping scheme Choose one of the available schemes to compute the hopping probabilities or turn off hopping.

Scaling and Damping These two prompts set the keywords **scaling** and **damping** in the SHARC input files. The scaling parameter has to be positive, and the damping parameter has to be in the interval [0, 1].

Atom masking In some cases, the script will ask to specify the atoms to which decoherence/rescaling/reflection should be applied. See section ?? for explanations (keyword **atommask**).

Gradient and nonadiabatic coupling selection For dynamics in the MCH representation, selection of gradients is used by default, and only one gradient (of the current state) is calculated. Selection of nonadiabatic couplings is only relevant if they are used (for propagation, gradient correction or rescaling of the velocities after a surface hop). For the selection threshold, usually 0.5 eV is sufficient, except if spin-orbit coupling is very strong and hence the gradients mix strongly. Sets the keywords **grad_select** and **nac_select** in the SHARC input files.

Laser file The user can specify to use an external laser field during the dynamics, and has to provide the path to the laser file (see section ?? and ??). **setup_traj.py** will check whether the number of steps and the time steps are compatible to the dynamics. If the interface can provide dipole moment gradients, **setup_traj.py** will also ask whether dipole moment gradients should be included in the simulations.

Dyson norm calculation If the interface is compatible, the user can request that Dyson norms are calculated on-the-fly. This option is only asked if Dyson norms can be computed (i.e., if states are present which differ by one electron, e.g., singlets and doublets).

THEODORE calculations If the interface is compatible, the user can request that THEODORE is run on-the-fly.

7.6.2 Interface-specific input

This input section is basically the same as for **setup_init.py** (section ?? and following sections). Note that for the dynamics simulations an initial wave function file is even more strongly recommended than for the initial excited-state calculations.

7.6.3 Output control

setup_traj.py will ask a number of questions regarding the content of the **output.dat** file, specifically about writing gradients, nonadiabatic coupling vectors, properties (1D and 2D), and overlap matrices to this file. Note that this is only possible if these quantities are actually calculated; otherwise, **sharc.x** will ignore these requestes.

7.6.4 Run script setup

Also this input section is very similar to the one in **setup_init.py** (see section ??).

7.6.5 Output

setup_traj.py will create for each initial state a directory where all trajectories starting in this state will be put. If the initial conditions file specified that the initial conditions are in the MCH representation, then the initial states will be assumed to be in the MCH representation as well. In this case, the directories will be named **Singlet_0**, **Singlet_1**, ..., **Doublet_0**, **Triplet_1**, ... If the initial states are in the diagonal representation, then the directories are simply called **X_1**, ... since they do not have a definite spin.

In each directory, subdirectories called **TRAJ_%05i** are created, where **%05i** is the initial condition index, padded to 5 digits with zeroes. In each trajectory's directory, an SHARC input file called **input** will be created, which contains all the dynamics options chosen during the **setup_traj.py** run. Also, files **geom** and **veloc** will be created. For trajectories setup with **setup_traj.py**, the determination of the initial wave function coefficients is done by SHARC. Furthermore, in each trajectory directory a subdirectory **QM** is created, where the **runQM.sh** script containing the call to the interface is put. In the directory **QM** also all interface-specific input files will be copied.

For each trajectory, a **run.sh** script will be created, which can be executed to run the dynamics simulation. You might need to adapt the run script to your cluster setup.

setup_traj.py also creates a script **all_run_traj.sh**, which can be used to execute all trajectories sequentially. Note that this is intended for small test trajectories, and should not be used for expensive production trajectories. For the

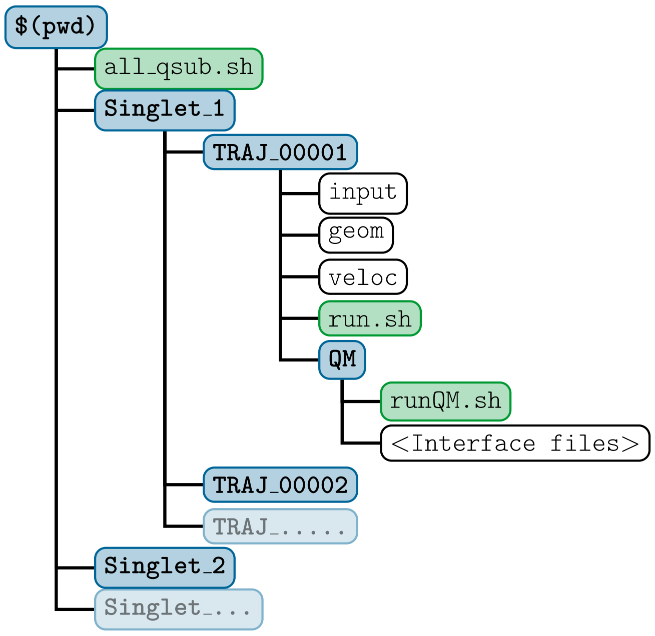

Figure 7.2: Directory structure created by **setup_traj.py**. Directories are in blue, executable scripts in green and regular files in black and white. Interface files usually include initial MO coefficients, template files and interface input files.

latter, **setup_traj.py** can optionally create a script **all_qsub_traj.sh**, which can be executed to submit all trajectories to a queueing system. You might need to adapt also this script to your cluster setup.

The full directory structure created by **setup_traj.py** is given in figure ??.

7.7 Laser field generation: laser.x

The Fortran code **laser.x** can generate files containing laser fields which can be used with SHARC. It is possible to superimpose several lasers, use different polarizations and apply a number of chirp parameters.

7.7.1 Usage

The program is simply called by

```
user@host> $SHARC/laser.x
```

It will interactively ask for the laser parameters. After input is complete, it writes the laser field to the file **laser** in the format which SHARC expects (see ??).

Similar to the interactive Python scripts, **laser.x** will also write the user input to **KEYSTROKES.laser**. After modifying this file, it can be used to directly execute **laser.x** without doing the interactive input again:

```
user@host> $SHARC/laser.x < KEYSTROKES.laser
```

7.7.2 Input

The first four options are global and need to be entered only once, all remaining input options need to be given for every laser pulse. For the definition of laser fields see section ??.

Number of lasers Any number of lasers can be used. The output file will contain the sum of all laser pulses defined.

Real-valued field If this is true, the output file will only contain the real parts of the laser field, while the columns defining the imaginary part of the field will be zero. Note, however, that SHARC will anyways only use the real part of the field in the simulations.

Time interval and steps The definitions of the starting time, end time and time step of the laser field must exactly match the simulation time and time substeps of the SHARC simulation. Note, that the laser field must always start at t=0 fs to be used with SHARC. The end time for the laser field must therefore coincide with the total simulation time given in the SHARC input. The number of time steps for the laser field is $t_{\text{total}}/\Delta t_{\text{sub}} + 1$.

Files for debugging This option is normally not needed, and can be set to False. If set to True, the chirped and unchirped laser fields in both time and frequency domain will be written to files called **`DEBUG_...`**.

Polarization vector The polarization vector **p** (will be normalized).

Type of envelope There are two options possible for the envelope function $\mathcal{E}(t)$, either a Gaussian envelope or a sinusoidal one (see **??**).

Field strength There are two input lines for the field strength $\mathcal{E}_0$, the first defining the unit in which the field strength is defined, the second gives the corresponding number. Field strength can be read in in GV/m, TW/cm^{-2} or atomic units.

FWHM and time intervals This option depends on the type of envelope chosen. While in both cases all 5 numbers need to be entered, for a Gaussian pulse only the first and third number have an effect. For a sinusoidal pulse all but the first number has an effect.

For a Gaussian pulse, the first argument corresponds to FWHM in equation (**??**) and the third argument to t_c in (**??**).

For a sinusoidal pulse, the second, third, fourth and fifth argument correspond to t_0, t_c, t_{c2} and t_e, respectively, in equation (**??**).

Central frequency There are two input lines for the central frequency ω_0. The first defines the unit (wavelength in nm, energy in eV, or atomic units). The second line gives the value.

Phase The total phase ϕ is given in multiples of π. For example, the input "**`1.5`**" gives a phase of $\frac{3\pi}{2}$.

Chirp parameters There are four lines giving the chirp parameters b_1, b_2, b_3 and b_4. See equation (**??**) for the meaning of these parameters.

7.8 Calculation of Absorption Spectra: `spectrum.py`

Aside from setting up trajectories, the **`initconds.excited`** files can also be used to generate absorption spectra based on the excitation energies and oscillator strengths in the file. The script **`spectrum.py`** calculates Gaussian, Lorentzian, or Log-normal convolutions of these data in order to obtain spectra. See section **??** for further details.

`spectrum.py` evaluates the absorption spectrum on a grid for all states it finds in an initial conditions file. Using command-line options, some initial conditions can be omitted in the convolution, see table **??**.

7.8.1 Input

The script is executed with the initial conditions file as argument:

```
user@host> $SHARC/spectrum.py [OPTIONS] initconds.excited
```

The script accepts a number of command-line options, which are given in table **??**.

Table 7.4: Command-line options for script **spectrum.py**.

Option	Description	Default
`-h`	Display help message and quit.	—
`-o FILENAME`	Output filename for the spectrum	**spectrum.out**
`-n INTEGER`	Number of grid points	500
`-e FLOAT FLOAT`	Energy range (eV) for the spectrum	1 to 10 eV
`-i INTEGER INTEGER`	Index range for the initial conditions	1 to 1000
`-f FLOAT`	FWHM (eV) for the spectrum	0.1 eV
`-G`	Gaussian convolution	Gaussian
`-L`	Lorentzian convolution	Gaussian
`-N`	Log-normal convolution	Gaussian
`-s`	Use only selected initial conditions	Use all
`-l`	Make a line spectrum	Convolution
`-D`	Compute density of states (ignore f_{osc})	Compute absorption
`--gnuplot FILENAME`	Write a GNUPLOT script	No GNUPLOT script
`-B INTEGER`	Perform **B** bootstrapping cycles (error estimation)	0
`-b FILENAME`	Output filename for bootstrapping	**spectrum_bootstrap.out**
`-r INTEGER`	Seed for random number generator (for bootstrap)	16661

7.8.2 Output

The script writes the absorption spectrum to a file (by default **spectrum.out**). Using the **-o** option, the user can redirect the output to a suitable file. The output is a table containing $n + 2$ columns, where n is the number of states found in the initial conditions file. The first column gives the energy in eV, within the given energy interval. In columns 2 to $n + 1$ the state-wise absorption spectra are given. The last column contains the total absorption spectrum, i.e., the sum over all states. The table has $n_{\mathrm{grid}} + 1$ rows. For line spectra the output format is exactly the same, however, the file will contain one row for each excited state of each initial condition in the initial conditions file. If density of states is computed, the script replaces the oscillator strength by a factor of 1 for all states.

Additionally, the script writes some information about the calculation to standard output, among these the maximum of the spectrum, which can be used in order to normalize the spectrum. The reported maximum is simply the largest value in the last column of the spectrum.

If requested, the script generates a GNUPLOT script, which can be used to directly plot the spectrum.

7.8.3 Error Analysis

The shape of the spectrum is strongly influenced by the number of initial conditions included and by the width of the broadening function (FWHM). In principle, the FWHM of the broadening function should be as small as possible and the number of initial conditions extremely large, in order to obtain a correctly sampled spectrum. In reality, if only few initial conditions were considered, the FWHM should be chosen large enough to smooth out any artifical structure of the spectrum arising solely from the small sample size.

In order to estimate whether the number of initial conditions and the FWHM are well-chosen, **spectrum.py** can compute error estimates for the total absorption spectrum. This estimate is computed by a bootstrapping procedure (similar to the one used in **bootstrap.py**). In order to use it, use the **-B** option with a positive integer argument (the default is zero, and hence no bootstrapping is performed). The procedure will generate a second output file, called **spectrum_bootstrap.out** by default. It contains in the first column the energy in eV, in the second the geometric average spectrum from all bootstrap cycles, in column 3 and 4 the positive and negative errors of the spectrum, and in all further columns the individual spectra obtained in the bootstrap cycles. In **gnuplot**, in order to plot the average spectrum and the upper and lower error bounds, plot **`u 1:2`**, **`u 1:($2+$3)`**, and **`u 1:($2+$4)`**.

A suitable procedure is to start with a rather small FWHM, compute the spectrum with errors, and if the errors are unsatisfactorily large, increase stepwise the FWHM. Note that the bootstrapping estimate will give very small errors if the FWHM is very large—even though the actual spectrum can look very different in this case.

7.9 File transfer: `retrieve.sh`

Usually, SHARC will run on some temporary directory, and not in the directory where the trajectories have been submitted from. The shell script **retrieve.sh** is a simple **scp** wrapper, which can be executed (in a directory where a trajectory has been sent from) in order to retrieve the output files of this trajectory. This might not work for every cluster setup.

It relies on the presence of the file **host_infos**. All trajectories set up with **setup_traj.py** create this file after the trajectory has been started with **run.sh**. **retrieve.sh** reads **host_infos** to determine the hostname and working directory of the trajectory and then uses **scp** to retrieve the output and restart files.

The script can be called with the option "**-lis**" in order to only retrieve the **output.lis** file, but not the other output files.

If the script is called with the option "**-res**" then also the restart files and the content of the **restart/** directory are copied.

It is advisable to configure public-key authentification for the hosts running the trajectories, so that not for every execution of **retrieve.sh** a password has to be entered.

7.10 Data Extractor: `data_extractor.x`

The **data_extractor.x** is the primary tool to extract useful, tabular data from the **output.dat** file that is produced by **sharc.x**. The produced files can then be further processed, e.g., by plotting them or by computing ensemble statistics with **data_collector.py** (section ??).

7.10.1 Usage

The **data_extractor.x** is a command line tool, and is called with the **output.dat** file as an argument, and possibly with some options.

```
user@host> $SHARC/data_extractor.x [options] output.dat
```

The program will create a directory **output_data/** in the current working directory (not necessarily in the directory where **output.dat** resides). In this directory, several files are written, containing, e.g., the potential energies depending on time, populations depending on time, etc. Which files are created can be controlled with the command line options, which are summarized in Table ??. For most applications, using the **-xs** or **-s** flags should be sufficient. The default is equivalent to **-s**. Note that some options might not be available if the necessary data is not written to **output.dat** (see write options in Table ??).

The program will extract the complete **output.dat** file until it reaches the EOF.

The **data_extractor.x** program will automatically detect the format of the **output.dat** file (the first SHARC release had a different file format than the more recent SHARC releases).

7.10.2 Output

After the program finishes, the directory **output_data/** contains a number of files. In each file, the number of columns is dependent in the total number n of states $i \in \{1...n\}$. The content of the files is listed in Table ??.

The file **expec.out** contains the information of **energy.out**, **spin.out** and **fosc.out** in one file. The content of **expec.out** can be conveniently plotted by using **make_gnuscript.py** (section ??) to generate a GNUPLOT script.

Table 7.5: Command-line options for **`data_extractor.x`**.

Option	Description	Default
`-h`	Display help message and quit.	—
`-f`	File name (can also be given without file name)	File name must be given
`-sk`	skip parsing of geom., vel., grad., NAC.	False
`-e`	Write **`energy.out`**	True
`-d`	Write **`fosc.out`**	True
`-da`	Write **`fosc_act.out`**	True
`-sp`	Write **`spin.out`**	True
`-cd`	Write **`coeff_diag.out`**, **`coeff_class_diag.out`**, **`coeff_mixed_diag.out`**	True
`-cm`	Write **`coeff_MCH.out`**, **`coeff_class_MCH.out`**, **`coeff_mixed_MCH.out`**	True
`-cb`	Write **`coeff_diab.out`**, **`coeff_class_diab.out`**, **`coeff_mixed_diab.out`**	True (if overlaps present)
`-p`	Write **`prob.out`**	True
`-x`	Write **`expec.out`**	True
`-xm`	Write **`expec_MCH.out`**	True
`-id`	Write **`ion_diag.out`**	False
`-im`	Write **`ion_MCH.out`**	False
`-dd`	Write **`dip_mom_diag.out`**	False
`-xs`	**`-e`**, **`,`** **`-d`**, **`,`** **`-cd`**, **`,`** **`-cm`**, **`,`** **`-p`**, **`,`** **`-x`**	
`-s`	**`-sp`**, **`,`** **`-xm`**, **`,`** **`-cb`**, **`,`** **`-da`** plus **`-xs`**	Default
`-l`	**`-id`**, **`,`** **`-im`** plus **`-s`**	
`-xl`	**`-dd`** plus **`-l`** (i.e., all)	

Table 7.6: Content of the files written by **`data_extractor.x`**. n is the total number of states, j is a state index ($j \in \{1..n\}$), and α is the active state.

File	# Columns	Columns	
`energy.out`	$4+n$	1	Time t (fs)
		2	Kinetic energy (eV)
		3	Potential energy (eV) of active state (diagonal)
		4	Total energy (eV)
		$4+j$	Potential energy (eV) of state j (diagonal)
`fosc.out`	$2+n$	1	Time t (fs)
		2	Oscillator strength from lowest to active state (diagonal)
		$2+j$	Oscillator strength from lowest state to state j (diagonal)
`fosc_act.out`	$1+2n$	1	Time t (fs)
		$1+j$	$E_j - E_{\text{active}}$ (in eV, diagonal)
		$1+n+j$	Oscillator strength from active state to state j (diagonal)
`spin.out`	$2+n$	1	Time t (fs)
		2	Total spin expectation value of active state
		$2+j$	Total spin expectation value of state j
`coeff_diag.out`	$2+2n$	1	Time t (fs)
		2	Norm of wave function $\sum_j \lvert c_j^{\text{diag}}\rvert^2$
		$1+2j$	$\Re(c_j^{\text{diag}})$
		$2+2j$	$\Im(c_j^{\text{diag}})$
`coeff_class_diag.out`	$2+n$	1	Time t (fs)
		2	1
		$2+j$	$\delta_{j\alpha}$
`coeff_mixed_diag.out`	$2+n$	1	Time t (fs)
		2	1
		$2+j$	$\delta_{j\alpha}$
`coeff_MCH.out`	$2+2n$	1	Time t (fs)
		2	Norm of wave function $\sum_j \lvert c_j^{\text{MCH}}\rvert^2$
		$1+2j$	$\Re(c_j^{\text{MCH}})$

Continued on next page

Table 7.6 – Continued from previous page

File	# Columns	Columns	
		$2+2j$	$\Im(c_j^{\mathrm{MCH}})$
`coeff_class_MCH.out`	$2+n$	1	Time t (fs)
		2	1
		$2+j$	$\lvert U_{j\alpha}\rvert^2$
`coeff_mixed_MCH.out`	$2+n$	1	Time t (fs)
		2	1
		$2+j$	$\lvert U_{j\alpha}\rvert^2+\sum_{k,l}2\Re(U_{jk}U_{jl}^* c_k^{\mathrm{diag}} c_l^{\mathrm{diag}*})$
`coeff_diab.out`	$2+2n$	1	Time t (fs)
		2	Norm of wave function $\sum_j \lvert c_j^{\mathrm{diab}}\rvert^2$
		$1+2j$	$\Re(c_j^{\mathrm{diab}})$
		$2+2j$	$\Im(c_j^{\mathrm{diab}})$
`coeff_class_diab.out`	$2+2n$	1	Time t (fs)
		2	1
		$2+j$	$\lvert T_{j\alpha}\rvert^2$
`coeff_mixed_diab.out`	$2+2n$	1	Time t (fs)
		2	1
		$2+j$	$\lvert T_{j\alpha}\rvert^2+\sum_{k,l}2\Re(T_{jk}T_{jl}^* c_k^{\mathrm{diag}} c_l^{\mathrm{diag}*})$
`prob.out`	$2+n$	1	Time t (fs)
		2	Random number from surface hopping
		$2+j$	Cumulated hopping probability $\sum_{k=1}^{j} P_k$
`expec.out`	$4+3n$	1	Time t (fs)
		2	Kinetic energy (eV)
		3	Potential energy (eV) of active state (diagonal)
		4	Total energy (eV)
		$4+j$	Potential energy (eV) of state j (diagonal)
		$4+n+j$	Total spin expectation value of state j (diagonal)
		$4+2n+j$	Oscillator strength of state j (diagonal)
`expec_MCH.out`	$4+3n$	1	Time t (fs)
		2	Kinetic energy (eV)
		3	Potential energy (eV) of approximate active state (MCH)
		4	Total energy (eV)
		$4+j$	Potential energy (eV) of state j (MCH)
		$4+n+j$	Total spin expectation value of state j (MCH)
		$4+2n+j$	Oscillator strength of state j (MCH)
`ion_diag.out`	$4+3n$	1	Time t (fs)
		$1+j$	$E_j - E_{\mathrm{active}}$ (in eV, diagonal)
		$1+n+j$	Dyson norm from active state to state j (diagonal)
`ion_MCH.out`	$4+3n$	1	Time t (fs)
		$1+j$	$E_j - E_{\mathrm{approximate\ active}}$ (in eV, mCH)
		$1+n+j$	Dyson norm from approximate active state to state j (MCH)
`dip_mom_diag.out`			Formatted like a minimal **`output.dat`** file containing the dipole moment matrices in diagonal representation.

7.11 Data Extractor for NetCDF: `data_extractor_NetCDF.x`

This program has the same function as the **`data_extractor.x`**. While the latter acts on ASCII-formatted **`output.dat`** files, the **`data_extractor_NetCDF.x`** reads only the header from **`output.dat`**, but the time step data from **`output.dat.nc`**. This file is obtained by setting **`output_format`** to **`ascii`** in the SHARC input file. For more details, see Section ??.

Note that this program currently does not support a few options of the **data_extractor.x**. In particular, the options **-sk**, **-dd**, **-id**, and **-im** are ignored and no corresponding output is produced.

Using the **-xyz** flag, the **data_extractor_NetCDF.x** can be used to write an **output.xyz** file from the NetCDF data file.

7.11.1 Usage

The **data_extractor_NetCDF.x** is used in the same way as **data_extractor.x**:

```
user@host> $SHARC/data_extractor_NetCDF.x [options] output.dat
```

Note that both **output.dat** and **output.dat.nc** need to be present. The file name of **output.dat.nc** is hardcoded, if your file is called differently, you should set a symbolic link.

7.11.2 Output

Same as **data_extractor.x**.

7.12 Data Converter for NetCDF: data_converter.x

This program can be used to convert an **output.dat** file into a NetCDF file (**output.dat.nc**). This significantly reduces the size of the file, and allows deleting the **output.xyz** file (its content will be in the NetCDF file).

7.12.1 Usage

The **data_converter.x** usage is very simple:

```
user@host> $SHARC/data_converter.x output.dat
```

Note that the **data_converter.x** does not modify the **output.dat** file. Hence, in order to save disk space, remove the data (below the header) afterwards:

```
user@host> sed -i ’1,/End of header array data/!d’ output.dat
```

7.12.2 Output

The program writes a file called **output.dat.nc**. This file can be extracted as usual with **data_extractor_NetCDF.x** (see section **??**).

7.13 Plotting the Extracted Data: make_gnuscript.py

The contents of the output files of **data_extractor.x** can be plotted with GNUPLOT. In order to quickly generate an appropriate GNUPLOT script, **make_gnuscript.py** can be used. The usage is:

```
user@host> $SHARC/make_gnuscript.py <S> [<D> [<T> [<Q> ... ] ] ]
```

make_gnuscript.py takes between 1 and 8 integers as command-line arguments, specifying the number of singlet, doublet, triplet, etc. states. It writes an appropriate GNUPLOT script to standard out, hence redirect the output to a file, e.g.:

```
user@host> $SHARC/make_gnuscript.py 3 0 2 > gnuscript.gp
```

Then, GNUPLOT can be run in the **output_data** directory of a trajectory:

```
user@host> gnuplot gnuscript.gp
```

This can also be accomplished in one step using a pipe, e.g.:

```
user@host> $SHARC/make_gnuscript.py 3 0 2 | gnuplot
```

The created plot script generates four different plots (press ENTER in the command-line where you started GNUPLOT to go to the next plot). The first plot shows the potential energy of all states in the dynamics over time in the diagonal representation. The currently occupied state is marked with black circles. A thin black line gives the total energy (sum of the kinetic energy and the potential energy of the currently occupied state). Each state is colored, with one color as contour and one color at the core of the line. The contour color represents the total spin expectation value of the state. The core color represents the oscillator strength of the state with the lowest state. See figure **??** for the relevant color code. Note that by definition the "oscillator strength" of the lowest state with itself is exactly zero, hence the lowest state is also light grey. This dual coloring allows for a quick recognition of different types of states in the dynamics, e.g. singlets vs. triplets or $n\pi^*$ vs. $\pi\pi^*$ states.

The second plot shows the same data again, but using relative energies such that the energy of the lowest state is zero.

The third plot shows the population $|c_i^{\mathrm{MCH}}|^2$ of the MCH electronic states over time. The line colors are auto-generated in order to give a large spread of all colors over the excited states, but the colors might be sub-optimal, e.g. for printing. In this cases, the user should manually adjust the colors in the generated script.

The fourth plot shows the population $|c_i^{\mathrm{diag}}|^2$ of the diagonal electronic states over time. These are the populations which are actually used for surface hopping. However, since these states are spin-mixed, it is usually difficult to interpret these populations.

The fifth plot shows the surface hopping probabilities over time. The plot is setup in such a way that the visible area corresponding to a certain state is proportional to the probability to hop into the state. Hence, if for a given time step the random number (black circles) lies within a colored area, a surface hop to the corresponding state is performed.

7.14 Ensemble Diagnostics Tool: `diagnostics.py`

The purpose of this script is to automatize the critical step of checking the trajectories in an ensemble for sanity before beginning the ensemble analysis.

The tool can check several different aspects of the trajectories. First, it checks whether all relevant output files of the trajectories are present, and if they are complete and consistent (e.g., no missing lines due to network/file system problems). Second, it checks simulation progress and status (e.g., whether the trajectory is running, crashed, finished, or stopped). Third, it can check several energy-related requirements: total energy conservation, smoothness of kinetic and potential energy, and hopping energy differences. It also checks for conservation of total population, and for trajectories using local diabatization also intruder states are checked.

Note that the diagnostics script can also be used to automatically run the **data_extractor.x** for all trajectories.

Also note that **diagnostics.py** will not work if the **printlevel** in the SHARC trajectories was lower than 2.

7.14.1 Usage

The script is interactive, simply start it with no command-line arguments or options:

```
user@host> $SHARC/diagnostics.py
```

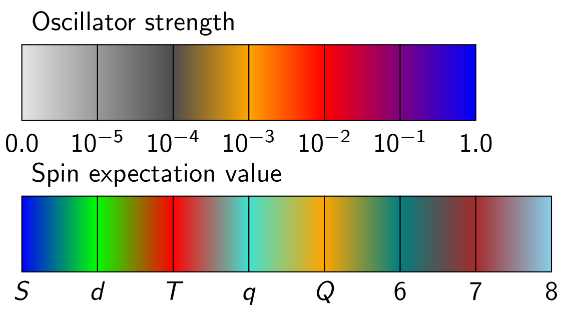

Figure 7.3: Color code for plots generated with the use of **make_gnuscript.py**.

7.14.2 Input

Paths to trajectories First the script asks the user to specify all directories for whose content the analysis should be performed. Enter one directory path at a time, and finish the directory input section by typing "**end**". Please do not specify each trajectory directory separately, but specify their parent directories, e.g. the directories **Singlet_1** and **Singlet_2**. **diagnostics.py** will automatically include all trajectories contained in these directories.

Unlike the ensemble analysis scripts (these are **populations.py**, **transition.py**, **crossing.py**, **trajana_essdyn.py**, **trajana_nma.py**, and **data_collector.py**, see below), **diagnostics.py** ignores files which indicate the status of a trajectory (**CRASHED**, **RUNNING**, **DONT_ANALYZE**) and carries out the diagnostics routines as long as it identifies a directory as a SHARC trajectory.

Settings The settings for the diagnostics run can be modified with a simple menu, which can be navigated with the commands **show**, **help**, **end**, and where the settings can be modified with **<key> <value>** (e.g., **hop_energy 0.2** sets the corresponding option to 0.2 eV). A list of the settings is given in Table ??.

Generally, the keywords **missing_output**, **missing_restart**, and **normal_termination** should always be left at **True**, since checking them is cheap and the obtained information is important. Note that **data_extractor.x** is always run for all trajectories, except if **output.dat** is older than the files in **output_data/**. During the check of each trajectory, **output.lis**, **output_data/energies.out**, and **output_data/coeff_diag.out** are furthermore checked for missing time steps.

Table 7.7: List of the settings for **diagnostics.py**.

Key	Value	Explanation
`missing_output`	Boolean	Checks if "output.lis", "output.log", "output.xyz", "output.dat" are existing. Setting to **False** only suppresses output, but files are always checked.
`missing_restart`	Boolean	Checks if "restart.ctrl", "restart.traj", "restart/" are existing. Files are not checked if set to **False**.
`normal_termination`	Boolean	Checks for status of trajectory: **RUNNING**: no finish message in **output.log**, last step started recently. **STUCK**: no finish message in **output.log**, last step started long ago. **CRASHED**: error message in **output.log**. **FINISHED**: finish message in **output.log**. **FINISHED (stopped by user)**: finished due to **STOP** file.
`etot_window`	Float	Maximum permissible drift (along full trajectory) in the total energy (in eV).
`etot_step`	Float	Maximum permissible total energy difference between two successive time steps (in eV).
`epot_step`	Float	Maximum permissible active state potential energy difference between two successive time steps (in eV). Not checked for time steps where a hop occurred.
`ekin_step`	Float	Maximum permissible kinetic energy difference between two successive time steps (in eV).
`pop_window`	Float	Maximum permissible drift in total population.
`hop_energy`	Float	Maximum permissible change in active state energy during a surface hop (in eV).
`intruders`	Boolean	Checks if intruder state messages in "output.log" refer to active state.
`always_update`	Boolean	Run the **data_extractor.x** always, even if **output.dat** is older than the produced files.
`extractor_mode`	String	Controls command line flags for the **data_extractor.x**: **xs**: Uses the **-xs** flag. **s**: Uses the **-s** flag. **l**: Uses the **-l** flag. **xl**: Uses the **-xl** flag. **dont**: **data_extractor.x** is never run (this leads to incomplete diagnostics, but is very fast).

Trajectory Flagging `diagnostics.py` determines for each trajectory a "maximum usable time" value (T_{mu}). This value is either the total simulation time or the time when the first violation (problems with time step consistency, total energy conservation, potential/kinetic energy smoothness, hopping energy restriction, or intruder states) in the trajectory appeared. The script prints the T_{mu} values for all trajectories at the end.

The user can then give a threshold for T_{mu}, so that **diagnostics.py** excludes all trajectories with values smaller than the threshold from analysis (the script will create a file **DONT_ANALYZE** in the directory of each affected trajectory). In this way it is possible to perform ensemble analysis for a given simulation length while ignoring problematic trajectories.

When choosing the threshold for T_{mu}, keep in mind that a compromise usually has to be made. A small value of the threshold will mean that many trajectories are admitted for analysis (because problems occurring late do not matter), giving good statistics, but that the analysis can only be carried out for the first part of the simulation time. On the other hand, choosing a large threshold allows analysis of a satisfactory simulation time, but only few trajectories will be included in the analysis (only the ones where no problems occurred for many time steps).

It is advisable that the chosen threshold value is used as input for the ensemble analysis scripts which ask for a maximum analysis time (**populations.py**, **transition.py**, **crossing.py**, **trajana_essdyn.py**, **trajana_nma.py**).

7.15 Calculation of Ensemble Populations: populations.py

For an ensemble of trajectories, usually one of the most relevant results are ensemble-averaged populations. The interactive script **populations.py** collects these populations from a set of trajectories.

Different methods to obtain populations or quantities approximating populations can be collected, as described below.

7.15.1 Usage

The script is interactive, simply start it with no command-line arguments or options:

```
user@host> $SHARC/populations.py
```

Depending on the analysis mode (see below) it might be necessary to run **data_extractor.x** for each trajectory prior to running **populations.py** (but **populations.py** can also call **data_extractor.x** for each subdirectory, if desired).

Paths to trajectories First, the script asks the user to specify all directories for whose content the analysis should be performed. Enter one directory path at a time, and finish the directory input section by typing "**end**". Please do not specify each trajectory directory separately, but specify their parent directories, e.g. the directories **Singlet_1** and **Singlet_2**. **populations.py** will automatically include all trajectories contained in these directories.

If you want to exclude certain trajectories from the analysis, it is sufficient to create an empty file called **CRASHED** or **RUNNING** in the corresponding trajectory directory. **populations.py** will ignore all directories containing one of these files. The file name is case insensitive, i.e., also files like **crashed** or even **cRasHED** will lead to the trajectory being ignored. Additionally, **populations.py** will ignore trajectories with a **DONT_ANALYZE** file from **diagnostics.py**.

Analysis mode Using **populations.py**, there are two basic ways in obtaining the excited-state populations. The first way is to count the number of trajectories for which a certain condition holds. For example, the number of trajectories in each classical state can be obtained in this way. However, it is also possible to count the number of trajectories for which the total spin expectation value is within a certain interval. The second way to obtain populations is to obtain the sum of the absolute squares of the quantum amplitudes over all trajectories. Table **??** contains a list of all possible analysis modes.

Run data extractor For analysis modes 6, 7, 8, 9 and 20 it is necessary to first run the data extractor (see section **??**). This task can be accomplished by **populations.py**. However, for a large ensemble or for long trajectories this may take some time. Hence, it is not necessary to perform this step each time **populations.py** is run.

populations.py will detect whether the file **output.dat** or the content of **output_data/** is more recent. Only if **output.dat** is newer the **data_extractor.x** will be run for this trajectory.

Note that mode 20 can only be used for trajectories using local diabatization propagation (keyword **coupling overlap** in SHARC input file) or .

Table 7.8: Analysis modes for **`populations.py`**. The last column indicates whether **`data_extractor.x`** has to be run prior to the ensemble analysis.

Mode	Description	From which file?	Extract?
1	For each diagonal state count how many trajectories have this state as active state.	`output.lis`	No
2	For each MCH state count how many trajectories have this state as approximate active state (see section **??**).	`output.lis`	No
3	For each MCH state count how many trajectories have this state as approximate active state (see section **??**). Multiplet components are summed up.	`output.lis`	No
4	Generate a histogram with definable bins (variable width). Bin the trajectories according to their total spin expectation value (of the currently active diagonal state).	`output.lis`	No
5	Generate a histogram with definable bins (variable width). Bin the trajectories according to their state dipole moment expectation value (of the currently active diagonal state).	`output.lis`	No
6	Generate a histogram with definable bins (variable width). Bin the trajectories according to the oscillator strength between lowest and currently active diagonal states.	`output_data/fosc.out`	Yes
7	Calculate the sum of the absolute squares of the diagonal coefficients for each state.	`output_data/coeff_diag.out`	Yes
8	Calculate the sum of the absolute squares of the MCH coefficients for each state.	`output_data/coeff_MCH.out`	Yes
9	Calculate the sum of the absolute squares of the MCH coefficients for each state. Multiplet components are summed up.	`output_data/coeff_MCH.out`	Yes
12	Transform option 1 to MCH basis (section **??**).	`output_data/coeff_class_MCH.out`	Yes
13	Transform option 1 to MCH basis (section **??**). Multiplet components are summed up.	`output_data/coeff_class_MCH.out`	Yes
14	Wigner-transform option 1 to MCH basis (section **??**).	`output_data/coeff_mixed_MCH.out`	Yes
15	Wigner-transform option 1 to MCH basis (section **??**). Multiplet components are summed up.	`output_data/coeff_mixed_MCH.out`	Yes
20	Calculate the sum of the absolute squares of the diabatic coefficients for each state (Only for trajectories with local diabatization).	`output_data/coeff_diab.out`	Yes
21	Transform option 1 to diabatic basis (section **??**).	`output_data/coeff_class_diab.out`	Yes
22	Wigner-transform option 1 to diabatic basis (section **??**). Multiplet components are summed up.	`output_data/coeff_mixed_diab.out`	Yes

Number of states For analysis modes 1, 2, 3, 7, 8 and 9 it is necessary to specify the number of states in each multiplicity. The number is auto-detected from the input file of one of the trajectories.

Intervals For analysis modes 4, 5 and 6 the user must specify the intervals (i.e., the histogram bins) for the classification of the trajectories. The user has to input a list of interval borders, e.g.:

```
Please enter the interval limits, all on one line.
Interval limits: 1e-3 0.01 0.1 1
```

Note that scientific notation can be used. Based on this input, for each time step a histogram is created with the number of trajectories in each interval. The histogram bins are:

1. $x \leq 10^{-3}$
2. $10^{-3} < x \leq 0.01$
3. $0.01 < x \leq 0.1$
4. $0.1 < x \leq 1$
5. $1 < x$

Note that there is always one more bin that interval borders entered.

Normalization If desired, **populations.py** can normalize the populations by dividing the populations by the number of trajectories.

Maximum simulation time This gives the maximum simulation time until which the populations are analyzed. For trajectories which are shorter than this value, the last population information is used to make the trajectory long enough. Trajectories which are longer are not analyzed to the end. **populations.py** prints the length of the shortest and longest trajectories after the analysis.

If **diagnostics.py** was executed previously, the user can enter here the threshold for the maximum usable time (see section ??).

Setup for bootstrapping The output file of **populations.py** is sufficient to perform kinetic model fits with the script **make_fitscript.py**. However, if error estimates for the kinetic model are desired (using **bootstrap.py**), the output file of **populations.py** is not enough. The user can tell **populations.py** to save additional data which is required by **bootstrap.py**.

Gnuplot script **populations.py** can generate an appropriate GNUPLOT script for the performed population analysis.

7.15.2 Output

By default, **populations.py** writes the resulting populations to **pop.out**. If the file already exists, the user is ask whether it shall be overwritten, or to provide an alternative filename. Note that the output file is checked only after the analysis is completed, so the program might run for a considerable amount of time before asking for the output file.

7.16 Calculation of Numbers of Hops: `transition.py`

Another important information from the trajectory ensemble is the number of hopping events and the involved states, for example to judge the relative importance of competing reaction pathways.

The interactive script **transition.py** calculates from an ensemble the number of hops between each pair of states and presents the results as “transition matrices”. Currently, the script employs the MCH active state information from **output.lis** for this computations. Note that since the MCH active state is only an approximate quantity (since hops are actually performed in the diagonal basis in SHARC), the results should be checked carefully. The script **transition.py** is still partly work-in-progress.

7.16.1 Usage

The script is interactive, simply start it with no command-line arguments or options:

```
user@host> $SHARC/transition.py
```

Paths to trajectories First the script asks the user to specify all directories for whose content the analysis should be performed. Enter one directory path at a time, and finish the directory input section by typing "**end**". Please do not specify each trajectory directory separately, but specify their parent directories, e.g. the directories **Singlet_1** and **Singlet_2**. **populations.py** will automatically include all trajectories contained in these directories.

If you want to exclude certain trajectories from the analysis, it is sufficient to create an empty file called **CRASHED** or **RUNNING** in the corresponding trajectory directory. **populations.py** will ignore all directories containing one of these files. The file name is case insensitive, i.e., also files like **crashed** or even **cRasHED** will lead to the trajectory being ignored. Additionally, **transition.py** will ignore trajectories with a **DONT_ANALYZE** file from **diagnostics.py**.

Analysis mode The different analysis modes for **transition.py** are given in Table **??**.

7.17 Fitting population data to kinetic models: make_fit.py

Often it is interesting to fit some functions to the population data from a trajectory ensemble, in order to provide a way to abstract the data and to obtain some kind of rate constants for population transfer, which allows to compare to experimental works. In simple cases, it might be sufficient to fit basic mono- and biexponential functions to the data, which provides the sought-after time constants. However, often a more meaningful approach is based on a chemical kinetics model. Such a model is specified by a set of chemical species (e.g., electronic states, reactants, products, etc) connected by elementary reactions with associated rate constants, which together form a reaction network graph. For an explanation of those graphs, see section **??**.

The script **make_fit.py** helps the user in implementing and fitting such global fits to a kinetics model.

The script works in a stand-alone fashion, unlike its predecessors (**make_fitscript.py** and **bootstrap.py**). It solves the differential equations numerically using a Runge-Kutta 5th order algorithm and fits the kinetic parameters using a number of different optimization algorithms. The script requires Python2 with NUMPY and SCIPY. If these are not available, use **make_fitscript.py** and **bootstrap.py**.

Often, one is also interested in obtaining an estimate of the error associated to these rate constants, e.g., in order to decide whether enough trajectories were computed. A possible way to obtain such error estimates is the statistical bootstrapping procedure. The idea of bootstrapping is to generate *resamples* of the original ensemble; for an ensemble of *n* trajectories, one draws *n* random trajectories with replacement to obtain one resample. The resample can then be fitted like the original ensemble to obtain a second estimate of the rate constants. By generating many resamples, one can thus obtain a "probability" distribution of the rate constants, from which a statistical error measure can be calculated. For details on these statistical measures, see section **??**.

The script **make_fit.py** implements this resampling–fitting–statistics procedure. It is dependent on the output of **populations.py**, but otherwise works in a stand-alone fashion.

Table 7.9: Analysis modes for **transition.py**. The last column indicates whether **data_extractor.x** has to be run prior to the ensemble analysis.

Mode	Description	From which file?	Extract?
1	Get transition matrix in MCH basis.	`output.lis`	No
2	Get transition matrix in MCH basis, ignoring hops within multiplets.	`output.lis`	No
3	Write a tabular file with the transition matrix in the MCH basis for each time step.	`output.lis`	No
4	Write a tabular file with the transition matrix in the MCH basis for each time step, ignoring hops within multiplets.	`output.lis`	No

7.17.1 Usage

The script is interactive, simply start it with no command-line arguments or options:

```
user@host> $SHARC/make_fit.py
```

Before you start the script, you need to prepare a file with the relevant populations data (usually, the output file of **populations.py** will suffice). You also might want to run **transition.py** first, which can help in developing a suitable kinetic model.

7.17.2 Input

The interactive input for this script consists of specifying the reaction network graph, the initial conditions and the data file. Additionally, the user has to specify which species should be fitted to which data columns in the file. Optionally, the user can specify the bootstrap settings for estimating errors.

Kinetic model species As a first step, the user has to specify the set of species in the model. Each species is fully described by its label. A label must start with a letter and can be followed by letters, numbers and underscores (although an underscore must not be directly followed by another underscore).

During input, the user can add one or several labels to the set of species, remove labels and display the current set of defined labels. It is not possible to add a label twice. Once all labels are defined, the keyword **end** brings the user to the next input section (hence, **end** is not a valid label).

Kinetic model elementary reactions Next, the reactions have to be defined. A reaction is specified by its initial species, final species, and reaction rate label. Reaction rate labels are under the same restrictions as species labels and must not be already used as a species label. Furthermore, initial and final species must be both defined previously and they must be different. There can only be one reaction from any species to another species (If a second reaction is defined, the first reaction label is simply overwritten). Note that reaction rate labels can be used in several reactions (in this way, different rates can be restricted to be the same).

During input, the user can add and remove reactions and display the currently defined reactions (displayed as a matrix). Unlike species labels, only one reaction can be added per line. Once all reactions are defined, the keyword **end** brings the user to the next input section.

Kinetic model initial values In order to specify the initial values for each species, the user simply has to define which species have non-zero initial population. These species will then be assigned an initial population constant, which can be fitted along with the reaction rates.

During input, the user can add and remove species from the set of species with non-zero initial population. Once all reactions are defined, the keyword **end** brings the user to the next input section.

Operation mode The script can either read a **pop.out** file for a simple global fit, or a **bootstrap_data/** directory for a global fit with error estimate. If the latter is chosen, the user needs to enter the number of bootstrap cycles.

Populations data file The user has to specify a path (autocomplete is enabled) to the file containing the population data to which the model functions should be fitted. In bootstrap mode, instead the path to the **bootstrap_data/** directory (can be prepared with **populations.py**) needs to be given. The script reads the file/files and automatically detects the maximum time and the number of data columns.

The file/files should be formatted as a table, with one time step per line (e.g., an output file of **populations.py**). On each line, the first number is interpreted as the time in femtoseconds and all consecutive numbers (separated by spaces) as the populations at this time. Note that the first entry must be at t=0 and all subsequent lines must be in strictly increasing order. Time steps can be unevenly spaced if necessary.

Species–Data mapping In the next setup section, the user has to specify which functions should be fitted to which data column from the data file. In the simplest case, one species is fitted to a single data column (e.g., the species **S0** is fitted to data column 2). However, it is also possible to fit the sum of two species to a column (this can be useful, e.g., to describe biexponential processes) and to fit a species to the sum of several columns (e.g., one can fit to the total triplet population to obtain a total ISC rate constant). In general, it is also possible to fit sums of species to sums of columns.

It is not allowed to use one species or one column in more than one mapping. However, it is possible to leave species or data columns unused in the global fit. While unused species still affect the outcome (through the reaction network definitions), unused data columns are simply ignored in the fit.

During input, the user can add one mapping per line as well as display the current mappings. For the column definitions, ranges can be given with the tilde symbol, e.g., **5~9** is interpreted as **5 6 7 8 9**. If a typo is made, the user can reset the mappings and repeat only the mapping input without the need to repeat the previous sections. Once all mappings are defined, the keyword **end** finishes the input section.

Fitting procedure In the last section, the user can edit the initial guesses for the rate constants and initial populations. To change a value, enter **label = value**. Use **show** to print the current values for all constants. **end** finishes the guess edit step.

Afterwards, the script asks whether the initial populations should be optimized or not. This is usually only useful if several species have non-zero initial populations. If you optimize initial populations, note that their sum might differ from 1 after optimization.

The script also asks whether the rate constants should be constrained to positive values. If answered with **yes**, then the optimized rate constants are restricted to the range 0.000 001 to infinity (i.e., the time constants are constrained between 1 000 000 fs and 0 fs). Note that with constraints SciPy uses the Trust Region Reflective algorithm, and the Levenberg-Marquardt algorithm for unconstrained cases.

7.17.3 Output

The script will write the output to standard out. It will first print the iterations of the fit, and the obtained results afterwards (all fitted parameters with errors). The script will also write two files, **fit_results.txt** and **fit_results.gp**. The latter is a GNUPLOT script that can be used to plot the global fit, which is useful for visual inspection.

Note that the errors printed for normal runs are just the intrinsic fitting errors, which assume that the population data is error-free. To obtain realistic fitting errors that take into account the uncertainty due to the finite trajectory ensemble, use the bootstrap mode.

In bootstrap mode, the script initially will perform the same steps as in normal mode, using the average population from the bootstrap directory. After writing the fit results and the two files, the script will start performing the bootstrap iterations, writing the fitted parameters in each iteration. In this way, the user can monitor the convergence of these values, to decide whether more iterations are required. Typically, the values and errors will vary strongly during the first iterations and stabilize later. The convergence rate is strongly dependent on the fitting model and the data.

After all iterations are done (or the script is interrupted with **Ctrl-C**), the script will print a summary of the statistical analysis. For each fitting parameter (all time constants and all initial populations), the script will list the arithmetic mean and standard deviation (absolute and relative), the geometric mean and standard deviation (separately for + and −, absolute and relative), and the minimum and maximum values. The script will also print a histogram with the obtained distribution for each parameter. For details on these statistical measures, see section ??.

bootstrap.py also creates an output file once it is finished. The file, **fit_bootstrap.txt**, contains the summary of the statistical analysis with the computed statistical measures and the histograms. Additionally, at the end this file contains a table with all obtained fitting parameters for resamples (e.g., for further statistical or correlation analysis).

7.18 Fitting population data to kinetic models: make_fitscript.py

This script has been superseded by **make_fit.py** (see section ??). You should use **make_fitscript.py** only if **make_fit.py** does not work.

Often it is interesting to fit some functions to the population data from a trajectory ensemble, in order to provide a way to abstract the data and to obtain some kind of rate constants for population transfer, which allows to compare to experimental works. In simple cases, it might be sufficient to fit basic mono- and biexponential functions to the data, which provides the sought-after time constants. However, often a more meaningful approach is based on a chemical kinetics model. Such a model is specified by a set of chemical species (e.g., electronic states, reactants, products, etc) connected by elementary reactions with associated rate constants, which together form a reaction network graph. For an explanation of those graphs, see section **??**.

The script **make_fitscript.py** helps the user in implementing and executing such global fits to a kinetics model.

The script employs the open-source computer algebra system MAXIMA as back-end to solve the differential equation systems which describe the model kinetics. The script writes a GNUPLOT script containing all commands necessary for the global fit. The fitting itself is performed with GNUPLOT.

7.18.1 Usage

The script is interactive, simply start it with no command-line arguments or options:

```
user@host> $SHARC/make_fitscript.py
```

Before you start the script, you need to prepare a file with the relevant populations data (usually, the output file of **populations.py** will suffice). You also might want to run **transition.py** first, which can help in developing a suitable kinetic model.

7.18.2 Input

The interactive input for this script consists of specifying the reaction network graph, the initial conditions and the data file. Additionally, the user has to specify which species should be fitted to which data columns in the file.

Kinetic model species As a first step, the user has to specify the set of species in the model. Each species is fully described by its label. A label must start with a letter and can be followed by letters, numbers and underscores (although an underscore must not be directly followed by another underscore).

During input, the user can add one or several labels to the set of species, remove labels and display the current set of defined labels. It is not possible to add a label twice. Once all labels are defined, the keyword **end** brings the user to the next input section (hence, **end** is not a valid label).

Kinetic model elementary reactions Next, the reactions have to be defined. A reaction is specified by its initial species, final species, and reaction rate label. Reaction rate labels are under the same restrictions as species labels and must not be already used as a species label. Furthermore, initial and final species must be both defined previously and they must be different. There can only be one reaction from any species to another species (If a second reaction is defined, the first reaction label is simply overwritten).

During input, the user can add and remove reactions and display the currently defined reactions (displayed as a matrix). Unlike species labels, only one reaction can be added per line. Once all reactions are defined, the keyword **end** brings the user to the next input section.

Kinetic model initial values In order to specify the initial values for each species, the user simply has to define which species have non-zero initial population. These species will then be assigned an initial population constant (e.g., for species **A** the initial population constant would be **A__0**). These constants will show up in the GNUPLOT script and their value can edited there. It is also possible to fit the initial populations to the data.

During input, the user can add and remove species from the set of species with non-zero initial population. Once all reactions are defined, the keyword **end** brings the user to the next input section.

Populations data file The user has to specify a path (autocomplete is enabled) to the file containing the population data to which the model functions should be fitted. The script reads the file and automatically detects the maximum time and the number of data columns.

The file should be formatted as a table, with one time step per line (e.g., an output file of **populations.py**). On each line, the first number is interpreted as the time in femtoseconds and all consecutive numbers (separated by spaces) as the populations at this time. Note that the first entry must be at t=0 and all subsequent lines must be in strictly increasing order. Time steps can be unevenly spaced if necessary.

Species–Data mapping In the final setup section, the user has to specify which functions should be fitted to which data column from the data file. In the simplest case, one species is fitted to a single data column (e.g., the species **S0** is fitted to data column 2). However, it is also possible to fit the sum of two species to a column (this can be useful, e.g., to describe biexponential processes) and to fit a species to the sum of several columns (e.g., one can fit to the total triplet population to obtain a total ISC rate constant). In general, it is also possible to fit sums of species to sums of columns.

It is not allowed to use one species or one column in more than one mapping. However, it is possible to leave species or data columns unused in the global fit. While unused species still affect the outcome (through the reaction network definitions), unused data columns are simply ignored in the fit.

During input, the user can add one mapping per line as well as display the current mappings. If a typo is made, the user can reset the mappings and repeat only the mapping input without the need to repeat the previous input. Once all mappings are defined, the keyword **end** finishes the input section.

Running MAXIMA Before the script attempts to call MAXIMA, it prints the command to be executed and asks the user for permission to execute. If permission is denied, the user may enter another command to call maxima. After permission is granted, the script calls MAXIMA. If the call takes more than a few seconds, the standard output of MAXIMA is printed and standard input is switched to MAXIMA. In this way, the user can answer any questions by MAXIMA (e.g., whether a constant/combination of constants is larger than zero). To answer these questions, type the answer followed by a semi-colon (e.g., **pos;**).

7.18.3 Output

After successful execution of the script, two files are created, **model_fit.dat** and **model_fit.gp**. The former file contains the populations data formatted appropriately for the global fit. The latter file contains the GNUPLOT script which can be executed by

```
user@host> gnuplot model_fit.gp
```

to perform the global fit. Please follow the instructions printed by **make_fitscript.py** at the end of the execution, in particular by adjusting the starting guesses for the fitting parameters (you have to open and edit **model_fit.gp** for this). Please do not change the structure of **model_fit.gp** if you intend to use this file with **bootstrap.py**.

7.19 Estimating Errors of Fits: bootstrap.py

This script has been superseded by **make_fit.py** (see section ??). You should use **bootstrap.py** only if **make_fit.py** does not work.

As was described in section ??, the population data from **populations.py** can be fitted to a kinetic model to obtain interpretable rate constants. Often, one is also interested in obtaining an estimate of the error associated to these rate constants, e.g., in order to decide whether enough trajectories were computed. A possible way to obtain such error estimates is the statistical bootstrapping procedure. The idea of bootstrapping is to generate *resamples* of the original ensemble; for an ensemble of *n* trajectories, one draws *n* random trajectories with replacement to obtain one resample. The resample can then be fitted like the original ensemble to obtain a second estimate of the rate constants. By generating many resamples, one can thus obtain a “probability” distribution of the rate constants, from which a statistical error measure can be calculated. For details on these statistical measures, see section ??.

The script **bootstrap.py** implements this resampling–fitting–statistics procedure. It is dependent on the output of **populations.py** and **make_fitscript.py**, and employs GNUPLOT to perform the actual fits.

7.19.1 Usage

The script is interactive, simply start it with no command-line arguments or options:

```
user@host> $SHARC/bootstrap.py
```

Before you start the script, you need to run **populations.py** to generate the bootstrapping data (**populations.py** asks the related question near the end of the input query).

Additionally, you need to use **make_fitscript.py** to generate a GNUPLOT fitting script. Please, do not modify the script (beyond initial values for the fitting constants), as **bootstrap.py** relies on the proper format of this file. It is advisable to create the GNUPLOT fitting script from one of the files contained in the bootstrap data directory (because then the simulation time is consistent between the fitting script and the bootstrapping data).

7.19.2 Input

The interactive input consists in specifying the paths to the bootstrapping data directory (from **populations.py**), the path to the fitting script, the number of resamples, and some other minor options.

Path to the bootstrapping data directory The user should enter here the path to the directory which was created by **populations.py** and which contains the bootstrapping raw data (i.e., the populations for each individual trajectory before summing up over the ensemble). **bootstrap.py** automatically detects the number of trajectories, time step, number of steps, and number of data columns.

Note that **bootstrap.py** only considers files in this directory if their filename starts with **pop_**. Also note that **bootstrap.py** creates a temporary subdirectory inside the bootstrapping data directory as scratch area.

Number of bootstrapping cycles Here, the number of resamples to generate and fit needs to be entered. The default, 10 resamples, is very small and not sufficient to achieve convergence in the errors for most applications. It is advisable to employ several hundred or thousand resamples to achieve good statistical convergence.

Note that the script can be interrupted during the iterations (using **Ctrl-C**) and then skips to the final analysis, so it is no problem to enter a too large number of cycles.

Random number generator seed The script employs random numbers to generate the random resamples. For reproducible results, do not use the default option (**!**), but enter an integer.

Path to the fitting script The user should enter the path to the appropriate GNUPLOT fitting script, generated with **make_fitscript.py**. It is strongly advisable to adjust the initial guesses for the fitting parameters in the script in order to ensure quick and stable convergence of the fits. Please, do not modify the generated GNUPLOT script in any other way, as this might break **bootstrap.py**.

Command to execute Here, the user should enter the command to be used to call GNUPLOT. In most cases, this will simply be **gnuplot**, but the user might want to use another installation of GNUPLOT for the fitting.

Number of CPUs **bootstrap.py** can be run in parallel mode, where multiple GNUPLOT processes are employed at the same time. Currently, this parallelization is not very efficient due to process creation overheads, but for complicated fitting scripts (where each fit takes relatively long), large data sets, or many resamples it can be useful.

Note that if more than one CPU is used, users should not prematurely terminate **bootstrap.py** with **Ctrl-C**, as some of the subprocesses might get stuck and the script will not properly go to the final analysis.

7.19.3 Output

During the resample iterations, **bootstrap.py** prints the current values and errors (geometric mean and geometric standard deviation) of the fitting parameters every few iterations. In this way, the user can monitor the convergence of these values, to decide whether more iterations are required. Typically, the values and errors will vary strongly during the first iterations and stabilize later. The convergence rate is strongly dependent on the fitting model and the data.

After all iterations are done (or the script is interrupted with **Ctrl-C**), the script will print a summary of the statistical analysis. For each fitting parameter (all time constants and all initial populations), the script will list the arithmetic mean and standard deviation (absolute and relative), the geometric mean and standard deviation (separately for + and −, absolute and relative), and the minimum and maximum values. The script will also print a histogram with the obtained distribution for each parameter. For details on these statistical measures, see section ??.

bootstrap.py also creates two output files. The first file, **bootstrap_cycles.out**, is written on-the-fly while the resample iterations are running. It contains the same tabular output as is shown in standard output, and can be used to visualize the convergence behavior of the parameters. For example, the first parameter can be monitored using GNUPLOT with this command: **plot "bootstrap_cycles.out" using 1:6:8 w yerrorbars**. The scond file, **bootstrap.out**, is written once the final analysis is complete. It contains the summary of the statistical analysis with the computed statistical measures and the histograms. Additionally, at the end this file contains a table with all obtained fitting parameters for resamples (e.g., for further statistical or correlation analysis).

7.20 Obtaining Special Geometries: crossing.py

In many cases, it is also important to obtain certain special geometries from the trajectories. The script **crossing.py** extracts geometries fulfilling special conditions from an ensemble of trajectories.

Currently, **crossing.py** finds geometries where the approximate MCH state (see section ??) of the last time step is different from the MCH state of the current time step (i.e. **crossing.py** finds geometries where surface hops occured).

7.20.1 Usage

The script is interactive, simply start it with no command-line arguments or options:

```
user@host> $SHARC/crossing.py
```

The input to the script is very similar to the one of **populations.py**.

Paths to trajectories First the script asks the user to specify all directories for whose content the analysis should be performed. Enter one directory path at a time, and finish the directory input section by typing “**end**”. Please do not specify each trajectory directory separately, but specify their parent directories, e.g. the directories **Singlet_1** and **Singlet_2**. **crossing.py** will automatically include all trajectories contained in these directories.

If you want to exclude certain trajectories from the analysis, it is sufficient to create an empty file called **CRASHED** or **RUNNING** in the corresponding trajectory directory. **crossing.py** will ignore all directories containing one of these files. Additionally, **crossing.py** will ignore trajectories with a **DONT_ANALYZE** file from **diagnostics.py**.

Analysis mode Currently, **crossing.py** only supports one analysis mode, where **crossing.py** is scanning for each trajectory the file **output.lis**. If the occupied MCH state (column 4 in output file **output.lis**) changes from one time step to the next, it is checked whether the old and new MCH states are the ones specified by the user. If this is the case, the geometry corresponding to the new time step (t) is retrieved from **output.xyz** (lines $t(n_{\text{atom}} + 2) + 1$ to $t(n_{\text{atom}} + 2) + n_{\text{atom}}$).

States involved in surface hop First, the user has to specify the permissible old MCH state. The state has to be specified with two integers, the first giving the multiplicity (**1**=singlet, ...), the second the state within the multiplicity (**1 1**=S_0, **1 2**=S_1, etc.). If a state of higher multiplicity is given, **crossing.py** will report all geometries where the old MCH state is any of the multiplet components.

For the new MCH state, the same is valid.

Third, the direction of the surface hop has to be specified. Choosing “Backwards” has the same effect as exchanging the old and new MCH states.

7.20.2 Output

All geometries are in the end written to an output file, by default **crossing.xyz**. The file is in standard xyz format. The comment of each geometry gives the path to the trajectory where this geometry was extracted, the simulation time and the diagonal and MCH states at this simulation time.

7.21 Internal Coordinates Analysis: `geo.py`

SHARC writes at every time step the molecular geometry to the file **output.xyz**. The non-interactive script **geo.py** can be used in order to extract internal coordinates from xyz files. The usage is:

```
user@host> $SHARC/geo.py [options] < Geo.inp > Geo.out
```

By default, the coordinates are read from **output.xyz**, but this can be changed with the **-g** option (see table **??**). Note that the internal coordinate specifications are read from standard input and the result table is written to standard out.

7.21.1 Input

The specifications for the desired internal coordinates are read from standard input. It follows a simple syntax, where each internal coordinate is specified by a single line of input. Each line starts with a one-letter key which specifies the type of internal coordinate (e.g. bond length, angle, dihedral, ...). The key is followed by a list of integers, specifying which atoms should be measured. As a simple example, **r 1 2** specifies the bond length (**r** is the key for bond lengths) between atoms 1 and 2. Note that the numbering of the atoms starts with 1. Each line of input is checked for consistency (whether any atom index is larger than the number of atoms, repeated atom indices, misspelled keys, wrong number of atom indices, ...), and erroneous lines are ignored (this is indicate by an error message).

Table **??** lists the available types of internal coordinates. The output is a table, where the first column is the time (Actually, the geometries are just enumerated starting with zero, and the number multiplied by the time step from the **-t** option). The successive columns in the output table list the results of the internal coordinates calculations. Each request generates at least one column, see table **??**.

Note that for most internal coordinates, the order of the atoms is crucial, since e.g. $a_{123} \neq a_{213}$. This also holds for the Cremer-Pople parameter requests. For these input lines, the atoms should be listed in the order they appear in the ring (clockwise or counter-clockwise).

Table 7.10: Possible types of internal coordinates in **geo.py**.

Key	Atom Indices	Description	Output columns
x	**a**	x coordinate of atom **a**	x
y	**a**	y coordinate of atom **a**	y
z	**a**	z coordinate of atom **a**	z
r	**a b**	Bond length between **a** and **b**	r
a	**a b c**	Angle between **a**–**b** and **b**–**c**	a
d	**a b c d**	Dihedral, i.e., angle between normal vectors of (**a,b,c**) and (**b,c,d**) (between -180° and 180°, same sign conventions as MOLDEN)	d
p	**a b c d**	Pyramidalization angle: 90° minus angle between bond **a**–**b** and normal vector of (**b,c,d**)	p
q	**a b c d**	Pyramidalization angle (alternative definition; angle between bond **a**–**b** and average of bonds **b**–**c** and **b**–**d**	q
5	**a b c d e**	Cremer-Pople parameters for 5-membered rings [?] and comformation classification.	q_2, ϕ_2, Boeyens
6	**a b c d e f**	Cremer-Pople parameters for 6-membered rings [?] and Boeyens classification [?].	Q, ϕ, θ, Boeyens
i	**a - f**	Angle between average plane through 3-rings (**a - c**) and (**d - f**)	i
j	**a - h**	Angle between average plane through 4-rings (**a - d**) and (**e - f**)	j
k	**a - j**	Angle between average plane through 5-rings (**a - e**) and (**f - j**)	k
l	**a - l**	Angle between average plane through 6-rings (**a - f**) and (**g - l**)	l
c		Writes the comment (second line of the xyz format) to the table.	Comment

As an advice, it is always a good idea to put the comment as the *last* request, if needed. Since the comment may contain blanks, having the comment not as the very last column might make it impossible to plot the resulting table.

The Boeyens classification symbols which are output for 6-membered rings are reported in LaTeX math code. Note that in the Boeyens classification scheme by definition a number of symbols are equivalent, and only one symbol is reported. These are the equivalent symbols: ${}^1C_4 = {}^3C_6 = {}^5C_2$, ${}^4C_1 = {}^6C_3 = {}^2C_5$, ${}^1T_3 = {}^4T_6$, ${}^2T_6 = {}^5T_3$, ${}^2T_4 = {}^5T_1$, ${}^3T_1 = {}^6T_4$, ${}^6T_2 = {}^3T_5$ and ${}^4T_2 = {}^1T_5$.

For 5-membered rings, the classification symbols are chosen similar to the Boeyens symbols. For the ${}^aE$ and E_a symbols, atom a is puckered out of the plane and the four other atoms are coplanar, while for the ${}^aH_b$ symbols the neighboring atoms a and b are puckered out of the plane in opposite directions and only the three remaining atoms are coplanar.

It is also possible to measure angles between the average planes through two n-membered rings. Currently, this is only possible if both rings have the same number of atoms (3, 4, 5, or 6).

7.21.2 Options

geo.py accepts a number of command-line options, see table **??**. All options have sensible defaults. However, especially if long comments should be written to the output file, it might be necessary to increase the field width. Note that the minimum column width is 20 so that the table header can be printed correctly. Also note that the column for the comment is enlarged by 50 characters.

Table 7.11: Command-line options for **geo.py**.

Option	Description	Default
`-h`	Display help message and quit.	—
`-b`	Report x, y, z, r, q_2 and Q in Bohrs	Angstrom
`-r`	Report a, d, p, q, ϕ_2, ϕ, θ, i, j, k, and l in Radians	Degrees
`-g FILENAME`	Read coordinates from the specified file	**output.xyz**
`-t FLOAT`	Assumed time step between successive geometries (fs)	1.0 fs
`-T INTEGER`	Start counting time step	0
`-w INTEGER`	Width of each column (min=20)	20
`-p INTEGER`	Precision (Number of decimals, min=width−3)	4

7.22 Essential Dynamics Analysis: trajana_essdyn.py

An essential dynamics analysis [?] is a procedure to find the most active vibrational modes in an ensemble of trajectories. It is based on the computation of the covariance matrix between all Cartesian (or mass-weighted Cartesian) coordinates of all steps of all trajectories and a singular value decomposition of this covariance matrix. For details on the computation, see section ??.

The interactive script **trajana_essdyn.py** can be used to perform such an analysis.

7.22.1 Usage

trajana_essdyn.py is an interactive script, which is started with:

```
user@host> $SHARC/trajana_essdyn.py
```

Note that before executing the script you should prepare an XYZ geometry file with the reference geometry (e.g., the ground state minimum or the average geometry from the trajectories).

7.22.2 Input

During interactive input, the script queries for the paths to the trajectories, the path to the reference structure, and a few other settings.

Path to the trajectories First the script asks the user to specify all directories for whose content the analysis should be performed. Enter one directory path at a time, and finish the directory input section by typing “**end**”. Please do not specify each trajectory directory separately, but specify their parent directories, e.g. the directories **Singlet_1** and **Singlet_2**. **trajana_essdyn.py** will automatically include all trajectories contained in these directories.

If you want to exclude certain trajectories from the analysis, it is sufficient to create an empty file called **CRASHED** or **RUNNING** in the corresponding trajectory directory. **trajana_essdyn.py** will ignore all directories containing one of these files. Additionally, **trajana_essdyn.py** will ignore trajectories with a **DONT_ANALYZE** file from **diagnostics.py**.

Path to reference structure For the essential dynamics analysis, a reference structure is required. This structure is substracted from all geometries for the correlation analysis. The structure can be in any format understandable by OPENBABEL, but the type of format needs to be specified. In most cases, the reference structure will be either in XYZ or MOLDEN format.

Mass-weighted coordinates If enabled, the correlation analysis will be carried out in mass-weighted Cartesian coordinates. In the output file, the mass-weighting will be removed properly.

Number of steps and time step These parameters are automatically detected and suggested as defaults. It is not necessary to change them.

Time step intervals By default, **trajana_essdyn.py** will analyze the full length of the simulations together. However, it is also possible to compute the essential dynamics for different time intervals (e.g., if the molecular motion is different in the beginning of the dynamics). Multiple, possibly overlapping, intervals can be entered; for each of the intervals, one set of output files is produced.

Results directory The path to the directory where the output files are stored. The path has to be entered as a relative or absolute path. If it does not exist, **trajana_essdyn.py** will create it.

7.22.3 Output

Inside the results directory, **trajana_essdyn.py** will create two subdirectories, **total_cov/** and **cross_av/**. In the directory **total_cov/** the results of the full covariance analysis are stored (i.e., essential modes found here have large total activity, but the trajectories could behave very differently). On the contrary, **cross_av/** will contain the results of the analysis of the average trajectory (i.e., essential modes found here have strongly coherent activity, where all trajectories behave similarly).

For each time step interval entered, one output file (e.g., **0-1000.molden**) is created in each of the two subdirectories.

7.23 Normal Mode Analysis: trajana_nma.py

A normal mode analysis [**?**] is a procedure to find the activity of given normal modes in an ensemble of trajectories For details on the computation, see section **??**.

The interactive script **trajana_nma.py** can be used to perform such an analysis.

7.23.1 Usage

trajana_nma.py is an interactive script, which is started with:

```
user@host> $SHARC/trajana_nma.py
```

Note that before executing the script you should prepare a MOLDEN file with the relevant normal modes.

7.23.2 Input

During interactive input, the script queries for the paths to the trajectories, the path to the normal mode file, and a few other settings.

Path to the trajectories First the script asks the user to specify all directories for whose content the analysis should be performed. Enter one directory path at a time, and finish the directory input section by typing "**end**". Please do not specify each trajectory directory separately, but specify their parent directories, e.g. the directories **Singlet_1** and **Singlet_2**. **trajana_nma.py** will automatically include all trajectories contained in these directories.

If you want to exclude certain trajectories from the analysis, it is sufficient to create an empty file called **CRASHED** or **RUNNING** in the corresponding trajectory directory. **trajana_nma.py** will ignore all directories containing one of these files. Additionally, **trajana_nma.py** will ignore trajectories with a **DONT_ANALYZE** file from **diagnostics.py**.

Path to normal mode file The normal mode file (in MOLDEN format) needs to contain the normal modes for which the analysis should be carried out. The file needs to have the same atom ordering as the trajectories.

Mass-weighted coordinates If enabled, the correlation analysis will be carried out in mass-weighted Cartesian coordinates. In the output file, the mass-weighting will be removed properly.

Number of steps and time step These parameters are automatically detected and suggested as defaults. It is not necessary to change them.

Automatic plot creation If available, the user can choose to automatically create total activity and temporal plots of the normal mode analysis.

Non-totally symmetric modes For symmetry reasons, the average value of a non-totally symmetric normal mode will always be close to zero for a sufficiently large ensemble. In order to still obtain useful information for such modes, the user can specify modes whose absolute value should be considered. Note that in the input it is possible to specify ranges, e.g., **2~8** for all modes from 2 to 8.

Multiplication by -1 The user can choose to multiply certain normal modes by -1. This is only for convenience when viewing the results. Note that in the input it is possible to specify ranges, e.g., **2~8** for all modes from 2 to 8.

Time step intervals By default, **trajana_nma.py** will analyze the full length of the simulations together. However, it is also possible to compute the essential dynamics for different time intervals (e.g., if the molecular motion is different in the beginning of the dynamics). Multiple, possibly overlapping, intervals can be entered; for each of the intervals, one set of output files is produced.

Results directory The path to the directory where the output files are stored. The path has to be entered as a relative or absolute path. If it does not exist, **trajana_nma.py** will create it.

7.23.3 Output

Inside the results directory, **trajana_nma.py** will create one subdirectory, **nma/**. By default, four output files are written into this subdirectory. The file **total_std.txt** will contain the mode activity for each normal mode and for each time interval; a large value indicates that across all time steps and trajectories there is a large variance in that normal mode. Similarly, the file **cross_av_std.txt** will contain the mode activity for each normal mode and for each time interval; this is computed from the average trajectory, therefore only showing coherent mode activity. The files **mean_against_time.txt** and **std_against_time.txt** contain for all normal modes and for all time steps the mean and standard deviation, respectively.

Also by default, additionally, inside each of the trajectory directories, **trajana_nma.py** generates three output files. The file **nma_nma.txt** is a table containing the normal mode coordinates of the trajectory for each time step; functionally, it is very similar to the output of **geo.py**. The files **nma_nma_av.txt** and **nma_nma_std.txt** contain for each time interval and each normal mode the mean and average for that trajectory.

Finally, if automatic plots were requested, the results subdirectory will contain two directories, **bar_graphs/** and **time_plots/**. The former contains plots of the data in **total_std.txt** and **cross_av_std.txt**, whereas the latter will contain plots of **mean_against_time.txt** and **std_against_time.txt**.

7.24 General Data Analysis: data_collector.py

Whereas most of the other analysis scripts in the SHARC suite are intended for rather specific tasks, **data_collector.py** is aimed at providing a general analysis tool to carry out a large variety of tasks. The primary task of **data_collector.py** is to collect tabular data from files which are present in all trajectories, possibly perform some analysis procedures (smoothing, averaging, convoluting, integrating, summing), and output the combined data as a single file. Possible applications of this functionality are statistical analysis of internal coordinates (e.g., mean and variation in a bond length), the creation of hair figures (e.g., a specific bond length plotted for all trajectories), data convolutions (e.g., distribution of bond length over time, simulation of time- and energy-resolved spectra), or data integration (e.g., computation of time-resolved intensities).

7.24.1 Usage

data_collector.py is an interactive script, which is started with:

```
user@host> $SHARC/data_collector.py
```

7.24.2 Input

In general, the first step in **data_collector.py** is to collect tabular data files which exist in the directories of multiple trajectories. For each trajectory, this file needs to have the same file name and the same tabular format; for example, one could read for all trajectories the **output.lis** files.

Collecting Hence, in the first input section, the script asks the user to specify all directories for whose content the analysis should be performed. Enter one directory path at a time, and finish the directory input section by typing "**end**". Please do not specify each trajectory directory separately, but specify their parent directories, e.g. the directories **Singlet_1** and **Singlet_2**. **data_collector.py** will automatically include all trajectories contained in these directories. If you want to exclude certain trajectories from the analysis, it is sufficient to create an empty file called **CRASHED** or **RUNNING** in the corresponding trajectory directory. **data_collector.py** will ignore all directories containing one of these files. Additionally, **data_collector.py** will ignore trajectories with a **DONT_ANALYZE** file from **diagnostics.py**.

In the second step, **data_collector.py** displays all files which appear in multiple trajectories and which might be suitable for analysis (the script ignores files which it knows to be not suitable, e.g., **output.dat**, **output.xyz**, most files in the **QM/** or **restart/** subdirectories, ...). All other files (e.g., **output.lis**, files in **output_data/**, ...) will be displayed, together with the number of appearances.

Once one of the files has been selected, one needs to assign the different data columns. (i) One column is designated the time column T, which defines the sequentiality of the data: (ii) Multiple columns can then be designated as data columns, called X columns in the following. (iii) The same number of columns is designated as weight columns, called Y columns here (weights can be set equal to 1 by selecting column "0" in the relevant menu). For example, for a time-resolved spectrum, the transition energies would be the X data, whereas the oscillator strengths would be the Y data (weights). With these assignments, the full data set is defined:

- For each trajectory a
  - For each time step t
    - For each X column i there will be a value pair $(x_i^a(t),y_i^a(t))$ or $(x_i^a(t),1)$.

In the simplest case, there will be exactly one $(x^a(t),1)$ pair for each trajectory and time, which is a two-dimensional data set. Keep in mind that in general, each trajectory could have different time steps at this point. We refer to this kind of data set (independent trajectories with possibly different time axes) as **Type1** data set. As will be described below, in **data_collector.py**, during certain processing steps the format of the data set is changed, which will create **Type2** or **Type3** data sets.

Once this data set is collected from the files (where too short or commented lines are ignored), **data_collector.py** allows for a number of subsequent processing steps, which are summarized in Figure ??.

Smoothing In this step, each trajectory is individually smoothed, using one of several smoothing kernels (Gaussian, Lorentzian, Log-normal, rectangular). Smoothing does not change the size or format of the data set, each value is simply

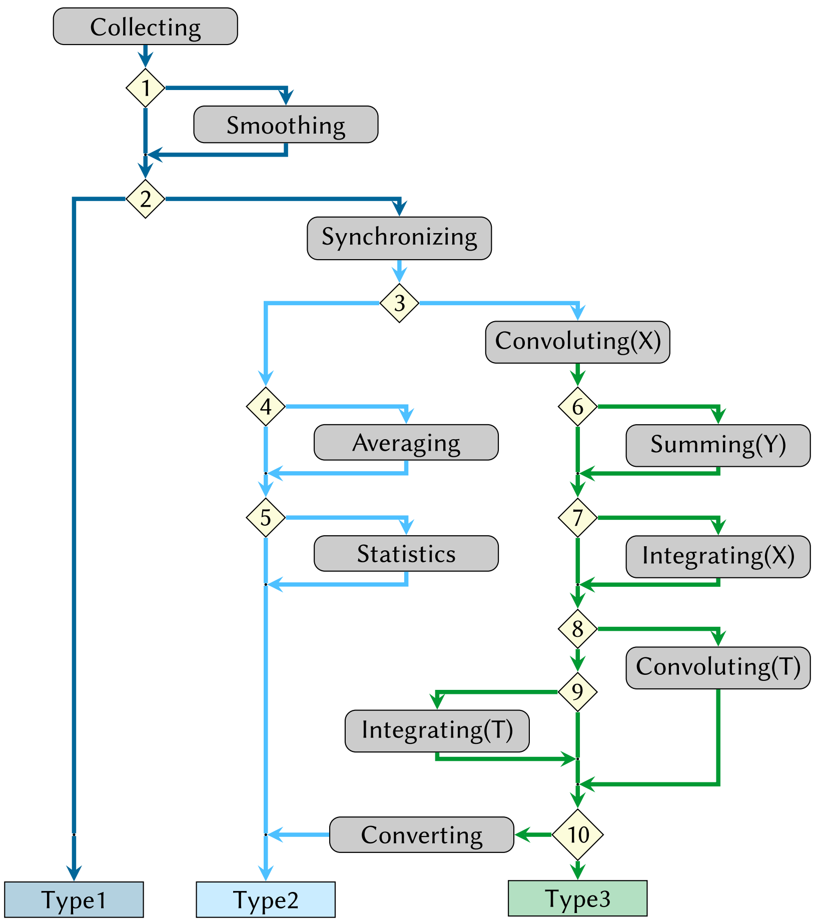

Figure 7.4: Possible workflows in **data_collector.py**. Grey boxes denote the different computational actions, yellow diamonds denote the different decisions which are queried in the input dialog, and the boxes at the denote the three different data set types (dark blue=**Type1**, light blue=**Type2**, green=**Type3**). The different actions and data set types are explained in the text.

replaced by the corresponding smoothed value; hence, a new **Type1** data set is obtained. Smoothing is applied to each X and each Y column independently, but always with the same kernel.

$$X_i^a(t) := \frac{\sum_{t'} X_i^a(t')f(t,t')}{\sum_{t'} f(t,t')} \quad \text{and analogously for } Y_i^a(t). \tag{7.1}$$

Here, $f(t,t')$ is the smoothing kernel.

Synchronizing In this step, the **Type1** data set is reformatted, by merging all trajectories together. This step creates a **Type2** data set, which has a common T axis for all trajectories (simply the union of the T columns from all trajectories). For each time step, all X and Y values of all trajectories are collected. If a trajectories does not have data at a particular time step, NaNs will be inserted. In this way, a rectangular, two-dimensional data set is obtained, with as many rows as time steps, and $2n_{\text{traj}}n_{\text{X}}$ data columns.

A simple application of Collecting+Synchronizing could be to generate a table with the bond length for all time steps for all trajectories, in order to generate a “hair figure”. This task could in principle also be accomplished with Bash tools like **awk** and **paste**, but this is troublesome if the trajectories are of different length, with different time steps, or if the table files contain comments.

Averaging The **Type2** data set from the Synchronizing step can contain a large number of data columns ($2n_{\text{traj}}n_{\text{X}}$). In order to reduce this amount of information, the Averaging step can be used to compute the mean and standard deviation across all trajectories, separately for each time step. This will create a new **Type2** data set, which still has a common time axis, but will only contain $4n_\mathrm{X} + 1$ data columns; these are the mean and standard deviation of all X and Y columns, plus one column giving the number of trajectories for each time step.

Currently, this step can be performed with either arithmetic mean/standard deviation or geometric mean/standard deviation.

Statistics Similar to the Averaging step, the Statistics step computes mean and standard deviations from a **Type2** data set. The difference is that during Statistics, these values are computed for all values from the first to the current time step. The data in the last time step thus gives the total statistics over all time steps and trajectories. The **Type2** data set from the Statistics step contains the same number of data columns as the one from the Averaging step.

Currently, this step can be performed with either arithmetic mean/standard deviation or geometric mean/standard deviation.

With the Averaging and Statistics steps, it is possible to compute the same data as with **trajana_nma.py** (if the appropriate files are read), namely the total (Statistics) and coherent (Averaging+Statistics) activity of the normal modes. Using **data_collector.py**, the same analysis can also be applied to internal coordinates computed with **geo.py**.

Convoluting(X) In order to create a data set which has common T *and* X axes (a **Type3** data set), it is in general necessary to perform some kind of data convolution involving the X column data. In order to do this, **data_collector.py** creates a grid along the X axis (n_grid points from the minimum value to the maximum value of all X data in the data set, plus some padding).

$$Y_i(t, X) := \sum_a Y_i^a(t) f(X, X_i^a(t)). \tag{7.2}$$

The created **Type3** data set has n_X data points for each time step and each X grid point.

Using energies as X columns and oscillator strengths/intensities as Y columns, in this way it is possible to compute time-dependent spectra.

Summing(Y) When n_X is larger than one, the Summing step can be used to compute the sum over all data points for each time step and each X grid point.

$$Y(t, X) := \sum_i Y_i(t, X). \tag{7.3}$$

This creates a new **Type3** data set, which will only contain one data point for each time step and each X grid point.

For example, for a transient spectrum involving multiple final states ($n_\mathrm{X} > 1$), after the Convoluting(X) step one obtains one transient spectrum for each final state. With the Summing(Y) step, one can then compute the total transient absorption spectrum.

Integrating(X) When computing transient spectra, one is often interested in integrating the spectrum within a certain energy window. This can be done with the Integrating(X) step. After entering a lower and upper X limit, a new **Type3** data set is created, with only three data points per time step. The first data point contains the integral from minus infinity to the lower limit, the second data point the integral between lower and upper limit, and the third data point the integral from the upper limit to infinity. If Summing(Y) was not carried out, this integration is carried out independently for each of the n_X data columns.

Convoluting(T) The **Type3** data set can also be convoluted along the time axis (e.g., in order to apply an instrument response function to a time-resolved spectrum). In order to do this, a uniform grid along the T axis is generated (with n_Tgrid points from the minimum value to the maximum value of the previous T axis, plus some padding).

$$Y_i(t', X) := \sum_t Y_i(t, X) f(t, t'). \tag{7.4}$$

The created **Type3** data set has as many data points for each X grid point as before, but the number of time steps is now n_Tgrid. Convoluting(T) can be applied also if Summing(Y) and/or Integrating(X) were used (in this case, the kernel is applied to the summed up or integrated data).

Integrating(T) This step carries out a cumulative summation along the T axis.

$$Y_i(t, X) := \sum_{t'=0}^{t} Y_i(t', X). \tag{7.5}$$

In this way, the data in the last time step constitute the integral over all time steps. Since all the partial cumulative sums are also computed, integrals within some bounds can simply be computed as differences between partial cumulative sums.

Converting At the end of the workflow, a **Type3** data set can be converted back into a **Type2** data set, which affects how the output file is formatted. This is usually a good idea if the Integrating(X) step was performed, but might not be a good idea otherwise. See below for how the different data set types are formatted on output.

7.24.3 Output

After the input dialog is finished, **data_collector.py** will start carrying out the requested analyses. For each of the workflow steps, one output file is written, so that all intermediate results can be used as well. Output files have automatically generated filenames, which describe how the data was obtained. Filenames are always in the form **collected_data_<T>_<X>_<Y>_<steps>.<type>.txt**, where **<T>** is an integer giving the T column index, **<X>** and **<y>** are lists of integers of the X and Y column indices, **<steps>** is a string denoting which workflow steps were carried out, and **<type>** denotes the data set type. For example, an output file could be named **collected_data_1_2_0_sy_cX.type3.txt**, where column 1 was the T column, column 2 the X column, no Y column was used, Synchronizing and Convoluting(X) were performed, resulting in a **Type3** data set.

Format of Type1 data set output **Type1** data sets are formatted such that each trajectory is given as a continuous block, separated by an empty line. Within each block, each line contains the data of one time step, order increasingly. Each line contains a trajectory index, the relative file path, the time, all X column data, and then all Y column data.

```
#    1                          2               3               4               5               6               7
#Index                   Filename            Time    X Column    5    X Column    6    Y Column    0    Y Column    0
     0 Singlet_1//TRAJ_00001/./Geo.out  0.00000000E+00  1.13340000E-01  7.99096900E+00  1.00000000E+00  1.00000000E+00
     0 Singlet_1//TRAJ_00001/./Geo.out  5.00000000E-01  1.59173000E-01  7.94395200E+00  1.00000000E+00  1.00000000E+00
     0 Singlet_1//TRAJ_00001/./Geo.out  1.00000000E+00  2.10868000E-01  7.89084000E+00  1.00000000E+00  1.00000000E+00
     ...

     1 Singlet_1//TRAJ_00002/./Geo.out  0.00000000E+00  5.03990000E-02  7.99078100E+00  1.00000000E+00  1.00000000E+00
     1 Singlet_1//TRAJ_00002/./Geo.out  1.00000000E+00  3.80370000E-02  8.00349700E+00  1.00000000E+00  1.00000000E+00
     1 Singlet_1//TRAJ_00002/./Geo.out  2.00000000E+00  1.09515000E-01  7.93073500E+00  1.00000000E+00  1.00000000E+00
     ...

     2 Singlet_1//TRAJ_00004/./Geo.out  0.00000000E+00  2.10908000E-01  8.29417600E+00  1.00000000E+00  1.00000000E+00
     2 Singlet_1//TRAJ_00004/./Geo.out  5.00000000E-01  1.49506000E-01  8.35651800E+00  1.00000000E+00  1.00000000E+00
     2 Singlet_1//TRAJ_00004/./Geo.out  1.00000000E+00  1.05887000E-01  8.40056700E+00  1.00000000E+00  1.00000000E+00
     ...
```

Note here in the example that the second trajectory has a time step of 1.0 fs and thus no data at 0.5 fs.

Format of Type2 data set output **Type2** data sets are formatted such that all trajectories share a common time axis, hence for each time step there will be one line of data. Each line starts with the time, followed columns with the data for the first trajectory, followed by the data for the second trajectory, etc. Within each trajectory, first all X columns, then all Y columns are given. If a Y column contains only unit weights (using special file column "0"), then this Y column is omitted from the **Type2** formatted output.

```
#            1               2               3               4               5               6               7 ...
#         Time    X Column    5    X Column    6    X Column    5    X Column    6    X Column    5    X Column    6  ...
 0.00000000E+00  1.13340000E-01  7.99096900E+00  5.03990000E-02  7.99078100E+00  2.10908000E-01  8.29417600E+00  ...
 5.00000000E-01  1.59173000E-01  7.94395200E+00             NaN             NaN  1.49506000E-01  8.35651800E+00  ...
 1.00000000E+00  2.10868000E-01  7.89084000E+00  3.80370000E-02  8.00349700E+00  1.05887000E-01  8.40056700E+00  ...
 ...
```

Note here in the example that the second trajectory does not have data at 0.5 fs.

Format of `Typ3` data set output **`Type3`** data sets are formatted such that all trajectories share common time and X axes. The data is formatted block-wise, with the first block corresponding to the first time step and containing all points on the X grid, followed by an empty line, followed by the second block, etc. Each block consists of n_{grid} lines, each starting with the time and X value in the first two columns and followed by n_{X} columns with the convoluted data.

```
#            1               2               3               4
#         Time          X_axis       Conv(5,0)       Conv(6,0)
 0.00000000E+00 -1.45534000E-01  2.38544715E-05  0.00000000E+00
 0.00000000E+00  2.30671958E-01  1.51462322E+00  0.00000000E+00
 0.00000000E+00  6.06877917E-01  1.07930050E-01  0.00000000E+00
 ...

 5.00000000E-01 -1.45534000E-01  1.31312692E-04  0.00000000E+00
 5.00000000E-01  2.30671958E-01  1.28614756E+00  0.00000000E+00
 5.00000000E-01  6.06877917E-01  4.46251462E-10  0.00000000E+00
 ...

 1.00000000E+00 -1.45534000E-01  8.75871124E-05  0.00000000E+00
 1.00000000E+00  2.30671958E-01  2.42291042E+00  0.00000000E+00
 1.00000000E+00  6.06877917E-01  1.60277894E-16  0.00000000E+00
 ...
```

7.25 Optimizations: `orca_External` and `setup_orca_opt.py`

All SHARC interfaces can deliver gradients for (multiple) ground and excited states in a uniform manner. This allows in principle to perform optimizations of excited-state minima, conical intersections, or crossing points. In order to employ a high-quality geometry optimizer for this task, the SHARC suite is interfaced to the external optimizer feature of ORCA. This is accomplished by providing the script **`orca_External`**, which is called by ORCA, runs any of the SHARC interfaces, constructs the appropriate gradient, and returns that to Orca. For the methodology used to construct the gradients, see section ??.

In order to easily prepare the input files for such an optimization, the script **`setup_orca_opt.py`** can be used. It takes a geometry file, interface input files, and the path to ORCA, and creates a directory containing all relevant input files. In the following, **`setup_orca_opt.py`** is described first, because it is the script which the user directly employs. Afterwards, **`orca_External`** is specified.

7.25.1 Usage

`setup_orca_opt.py` is an interactive script, which is started with:

```
user@host> $SHARC/setup_orca_opt.py
```

Note that before executing the script you should prepare a template for the interface you want to use (as, e.g., in **`setup_init.py`** or **`setup_traj.py`**).

7.25.2 Input

In the input section, the script asks for: (i) the path to ORCA, (ii) the input geometries, (iii) the optimization settings, (iv) the interface settings.

Path to ORCA Here the user is prompted to provide the path to the ORCA directory. Note that the script will not expand the user (~) and shell variables (since possibly the calculations are running on a different machine than the one used for setup). ~ and shell variables will only be expanded during the actual calculation.

Interface In this point, choose any of the displayed interfaces to carry out the ab initio calculations. Enter the corresponding number.

Input geometry Here the user is prompted for a geometry file in XYZ format, containing one or more geometries (with consistent number of atoms). For each geometry in this file, a directory with all input files is created, in order to carry out multiple optimizations (e.g., with output from **crossing.py**).

Number of states Here the user can specify the number of excited states to be calculated. Note that the ground state has to be counted as well, e.g., if 4 singlet states are specified, the calculation will involve the S_0, S_1, S_2 and S_3. Also states of higher multiplicity can be given, e.g. triplet or quintet states.

States to optimize Two different optimization tasks can be carried out: optimization of a minimum (ground or excited state) or optimization of a crossing point (either a conical intersection between states of the same multiplicity or a minimum-energy crossing points between states of different multiplicities; this is detected automatically).

For minima, the state to optimize needs to be specified. For crossing points, the two involved states need to be specified. In all cases, the specified states need to be included in the number of states given before.

CI optimization parameters If you are optimizing a conical intersection (states of same multiplicity) with an interface which cannot provide nonadiabatic coupling vectors (e.g., **SHARC_RICC2.py**, **SHARC_ADF.py**, or **SHARC_GAUSSIAN.py**), then the optimization will employ the the penalty function method of Levine, Coe, and Martínez [?]. In this method, a penalty function is optimized, which depends on the energies of the two states and on two parameters, σ and α (see section ?? for their mathematical meaning).

Practically, the parameters affect how close the minimum of the penalty function is to the true minimum on the crossing seam and how hard the optimization will be to converge. Generally, a large σ will allow going closer to the true conical intersection, but will make the penalty function more stiff (steeper harmonic) and thus harder to optimize. A small α will also allow going closer to the true conical intersection, but will make the penalty function less harmonic; at $\alpha = 0$, the penalty function will have a cusp at the minimum, making it unoptimizable because the gradient never becomes zero.

The default values, 3.5 and 0.02, are the ones suggested in [?]. They can be regarded as relatively soft, i.e., they enable a very smooth convergence but might lead to unacceptably large energy gaps at convergence (i.e., the minimum of the penalty function is too far from the true minimum of the crossing seam). In this case, it is advisable to restart the optimization from the last point with increased σ (e.g., by a factor of 4), and simultaneously reducing the maximum step (see next point). The α is best left at the suggested value of 0.02 to avoid the cusp problem.

Maximum allowed displacement Within ORCA, it is possible to restrict the maximum step (the trust radius) of the optimizer. A larger maximum step might decrease the number of iterations necessary, but might also lead to instabilities in the optimization (if the potential energy surface is very steep or anharmonic). Hence, it can be advisable to reduce the maximum allowed step (from the default of 0.3 a.u.), especially if the starting geometry is already very good (e.g., after restart with increase of σ) or if the potential is known to be stiff (strong bond, large σ, small α, ...). Note that the maximum step can be restricted even the penalty function method is not used and σ and α are not relevant.

Interface-specific input This input section is basically the same as for **setup_init.py** (sections ??). Also for the optimizations an initial wave function file should be provided, especially for the multi-reference methods.

Run script setup Here the user needs to provide the path to the directory where the optimizations should be setup.

7.25.3 Output

Inside the specified directory, **setup_orca_opt.py** creates one subdirectory for each geometry in the input geometry file. Each subdirectory is prepared with the corresponding initial geometry file (**geom.xyz**), the ORCA input file (**orca.inp**), the appropriate interface-specific files (template, resources, QM/MM), and a shell script for execution (**run_EXTORCA.sh**).

In order to run one of the optimizations, execute the shell script or send it to a batch queuing system. Note that $SHARC needs to be added to the $PATH so that ORCA can find **orca_External** (this is automatically done inside **run_EXTORCA.sh**).

When the shell script is started, ORCA will write a couple of output files, where the two most relevant are **orca.trj** and **orca.log**. The former is an XYZ file with all geometries from the optimization steps. The latter (the ORCA standard output) contains all details of the optimization (convergence, step size, etc) as well as a summary of what **orca_External** did (after the line **EXTERNAL SHARC JOB**). This summary contains all relevant energies and shows how the gradient is constructed. Note that in each iteration, a line starting with **>>** is written, which contains the energies of the optimized state(s). This line can easily be extracted with **grep** to follow the optimization of a crossing point.

7.25.4 Description of orca_External

orca_External provides a connection between the external optimizer of ORCA and any of the SHARC interfaces. In figure ??, the file communication between ORCA, **orca_External**, and the interfaces is presented.

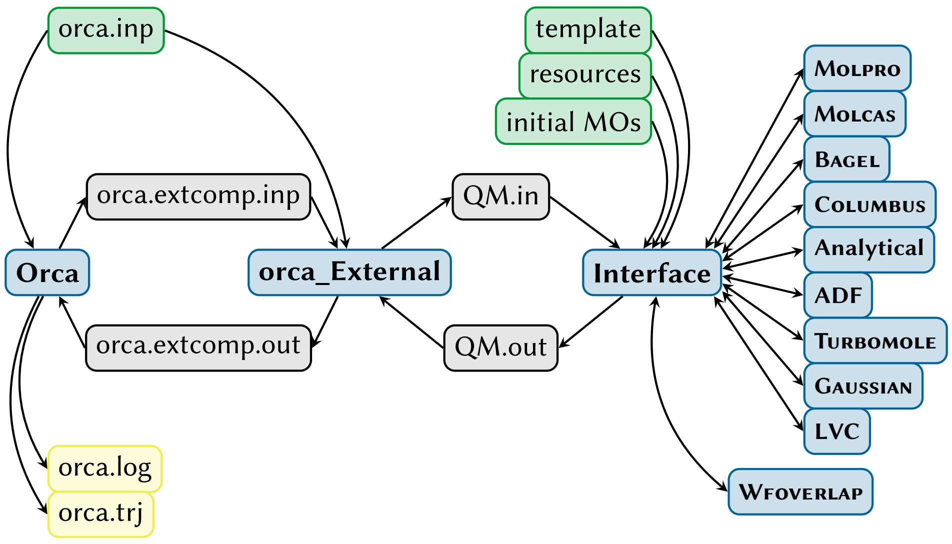

Figure 7.5: Communication between ORCA, **orca_External**, the interfaces, and the quantum chemistry codes.

As can be seen, **orca_External** writes **QM.in** and **QM.out** files, in the same way that **sharc.x** is doing. All information to write the **QM.in** file comes from the ORCA communication file **orca.extcomp.inp** (geometry) and the ORCA input file (number of states, interface, states to optimize). To provide the latter information, **orca_External** reads specially marked comments from the ORCA input file which are ignored by ORCA. These comments start with **#SHARC:**, followed by a keyword (**states**, **interface**, **opt**, or **param**) and the keyword arguments.

7.26 Single Point Calculations: setup_single_point.py

It is possible to run single point calculations through the SHARC interfaces. This is useful, e.g., to do a computation in exactly the same way as during the dynamics simulations. Single point calculations using the interfaces can also be easily automatized.

7.26.1 Usage

setup_single_point.py is an interactive script, which is started with:

```
user@host> $SHARC/setup_single_point.py
```

Note that before executing the script you should prepare a template for the interface you want to use (as, e.g., in **setup_init.py** or **setup_traj.py**).

7.26.2 Input

In the input section, the script asks for: (i) the input geometries, (ii) the number of states, and (iii) the interface settings.

Interface First, choose any of the displayed interfaces to carry out the ab initio calculations. Enter the corresponding number.

Input geometry Here the user is prompted for a geometry file in XYZ format, containing one or more geometries (with consistent number of atoms). For each geometry in this file, a directory with all input files is created, in order to carry out multiple optimizations (e.g., with output from **crossing.py**).

Number of states Here the user can specify the number of excited states to be calculated. Note that the ground state has to be counted as well, e.g., if 4 singlet states are specified, the calculation will involve the S_0, S_1, S_2 and S_3. Also states of higher multiplicity can be given, e.g. triplet or quintet states.

Interface-specific input This input section is basically the same as for **setup_init.py** (sections **??**). Also for the optimizations an initial wave function file should be provided, especially for the multi-reference methods.

Run script setup Here the user needs to provide the path to the directory where the optimizations should be setup.

7.26.3 Output

Inside the specified directory, **setup_single_point.py** creates one subdirectory for each geometry in the input geometry file. Each subdirectory is prepared with the corresponding geometry file (**QM.in**), the appropriate interface-specific files (template, resources, QM/MM), and a shell script for execution (**run.sh**).

In order to run one of the optimizations, execute the shell script or send it to a batch queuing system.

When the shell script is started, the chosen SHARC interface is executed. The interface writes a file called **QM.log** which contains details of the computation and progress status. The final results of the computation are written to **QM.out**, which can be inspected manually. Alternatively, some basic data (excitation energies, oscillator strengths) can be computed with **QMout_print.py** (section **??**).

7.27 Format Data from QM.out Files: QMout_print.py

With the script **QMout_print.py** one can print a table with energies and oscillator strengths from a **QM.out** file, as it is produced by the interfaces.

7.27.1 Usage

QMout_print.py is a command line tool, and is executed like this:

```
user@host> $SHARC/QMout_print.py [options] QM.out
```

The options are summarized in Table **??**

Table 7.12: Command-line options for **QMout_print.py**.

Option	Description	Default
`-h`	Display help message and quit.	—
`-i FILENAME`	Path to **QM.in** file (to read number of states)	—
`-s INTEGERS`	List of numbers of states per multiplicity	1
`-e FLOAT`	Absolute energy shift in Hartree	0.0
`-D`	Output diagonal states	MCH states

7.27.2 Output

The script prints a table with state index, state label, energy, relative energy, oscillator strength, and spin expectation value to standard output.

7.28 Diagonalization Helper: `diagonalizer.x`

The small program **`diagonalizer.x`** is used by the Python scripts to diagonalize matrices if the NUMPY library is not available. Currently, only **`excite.py`** and **`SHARC_Analytical.py`** need to diagonalize matrices. The program **`diagonalizer.x`** is implemented as a simple front-end to the LAPACK libary.

The program reads from stdin. The first line consists of the letter "r" or "c", followed by two integers giving the matrix dimensions. For "r", the program assumes a real symmetric matrix, for "c" a Hermitian matrix. The matrix must be square. The second line is a comment and is ignored. In the following lines, the matrix elements are given. On each line one row of the matrix has to be written. If the matrix is complex, each line contains the double number of entries, where real and imaginary part are always given subsequently. The following is an example input:

```
c 2 2
comment
1.0  0.0    2.0  0.1
2.0 -0.1    6.0  0.0
```

for the matrix

$$A = \begin{pmatrix} 1 & 2 + 0.1i \\ 2 - 0.1i & 6 \end{pmatrix}. \tag{7.6}$$

The diagonal matrix and the matrix of eigenvectors is written to stdout.

8 Methods and Algorithms

In this chapter different aspects of Sharc simulations are discussed in detail. The topics are ordered alphabetically.

8.1 Absorption Spectrum

Using **spectrum.py**, an absorption spectrum can be calculated as the sum over the absorption spectra of each individual initial condition:

$$\sigma(E) = \sum_i^{n_\text{init}} \sigma_i(E), \tag{8.1}$$

where i runs over the initial conditions.

The spectrum of a single initial condition is the convolution of its line spectrum, defined through a set of tuples $(E^\alpha, f^\alpha_\text{osc})$ for each electronic state α, where E^α is the excitation energy and $f^\alpha_\text{osc})$ is the oscillator strength.

The convolution of the line spectrum can be performed with **spectrum.py** using either Gaussian or Lorentzian functions. The contribution of a state α to the absorption spectrum $\sigma^\alpha(E)$ is given by:

$$\sigma^\alpha_\text{Gaussian}(E) = (f_\text{osc})^\alpha_i \, \mathrm{e}^{c\left(E-E^\alpha_i\right)^2}, \tag{8.2}$$

$$\text{with} \qquad c = -\frac{4\ln(2)}{\text{FWHM}^2}, \tag{8.3}$$

or

$$\sigma^\alpha_\text{Lorentzian}(E) = \frac{(f_\text{osc})^\alpha_i}{\frac{1}{c}\left(E-E^\alpha_i\right)^2+1}, \tag{8.4}$$

$$\text{with} \qquad c = \frac{1}{4}\text{FWHM}^2, \tag{8.5}$$

or

$$\sigma^\alpha_\text{Log-normal}(E) = (f_\text{osc})^\alpha_i \, \frac{E^\alpha_i}{E} \mathrm{e}^{-\frac{c}{4\ln(2)}-\frac{\ln(2)}{c}\left(\ln(E)-\ln(E^\alpha_i)\right)^2}, \tag{8.6}$$

$$\text{with} \qquad c = \left[\ln\left(\frac{\text{FWHM}+\sqrt{\text{FWHM}^2+4(E^\alpha_i)^2}}{2E^\alpha_i}\right)\right]^2, \tag{8.7}$$

where FWHM is the full width at half maximum.

8.2 Active and inactive states

Sharc allows to “freeze” certain states, which then do not participate in the dynamics. Only energies and dipole moments are calculated, but all couplings are disabled. In this way, these states are never visited (hence also no gradients and nonadiabatic couplings are calculated, making the inclusion of these states cheap). Example:

```
nstates 2 0 2
actstates 2 0 1
```

In the example given, state T_2 is frozen. The corresponding Hamiltonian looks like:

$$\mathbf{H} = \begin{pmatrix} E(S_0) & & a_{01}^* & a_{02}^* & b_{01}^* & b_{02}^* & a_{01} & a_{02} \\ & E(S_1) & a_{11}^* & a_{12}^* & b_{11}^* & b_{12}^* & a_{11} & a_{12} \\ a_{01} & a_{11} & E(T_1) & p_{12}^* & & -q_{12}^* & & \\ a_{02} & a_{12} & p_{12} & E(T_2) & q_{12}^* & & & \\ b_{01} & b_{11} & & -q_{12} & E(T_1) & & & -q_{12}^* \\ b_{02} & b_{12} & q_{12} & & & E(T_2) & q_{12}^* & \\ a_{01}^* & a_{11}^* & & & & -q_{12} & E(T_1) & p_{12} \\ a_{02}^* & a_{12}^* & & & q_{12} & & p_{12}^* & E(T_2) \end{pmatrix} \tag{8.8}$$

where all matrix elements marked red are deleted, since T_2 is frozen.

The corresponding matrix elements are also deleted from the nonadiabatic coupling and overlap matrices. For propagation including laser fields, also the corresponding transition dipole moments are neglected, while the transition dipole moments still show up in the output (in order to characterize the frozen states).

Active and frozen states are defined with the **states** and **actstates** keywords in the input file. Note that only the highest states in each multiplicity can be frozen, i.e., it is not possible to freeze the T_1 while having T_2 active. However, it is possible to freeze all states of a certain multiplicity.

8.3 Amdahl's Law

Some of the interfaces (**SHARC_MOLCAS.py**, **SHARC_ADF.py** **SHARC_GAUSSIAN.py**, **SHARC_ORCA.py**) use Amdahl's law to predict the most efficient way to run multiple calculations in parallel, where each calculation itself is parallelized. For example, in **SHARC_GAUSSIAN.py** it might be necessary to compute the gradients of five states, using four CPU cores. The most efficient way to run these five jobs depends on how well the GAUSSIAN computation scales with the number of cores—for bad scaling, running four jobs on one core each followed by the fifth job might be best, whereas for good scaling, running each job on four cores subsequently is better because no core is idle at any time. In order to automatically distribute the jobs efficiently, the interfaces use Amdahl's law, which can be stated as:

$$T(n_{\text{core}}) = T(1)\left(1 - r + \frac{r}{n_{\text{core}}}\right). \tag{8.9}$$

Here, $T(1)$ is the run time of the job with one CPU core, and r is the fraction of $T(1)$ which benefits from parallelization. The parameter r can be given to the interfaces; it is between 0 and 1, where 0 means that the calculation does not get faster at all with multiple cores, whereas 1 means that the run time scales linearly with the number of cores.

8.4 Bootstrapping for Population Fits

Bootstrapping, in the context of population fitting, is a statistical method to obtain the statistical distribution of the fitted parameters, which can be used to infer the error associated with the fitted parameter. The general idea of bootstrapping is to take the original sample (the set of trajectories in the ensemble), and generate new samples (resamples) by randomly drawing trajectories "with replacement" from the original ensemble. These resamples will differ form the original ensemble by containing some trajectories multiple times while other trajectories might be missing. For each of the resamples, the fitting parameter are obtained normally and saved for later analysis.

After many resamples, we obtain a list of many "alternative" parameters, which can be plotted in a histogram to see the statistical distribution of the fitting parameter. The number of resamples should generally be large (several hundred or thousand resamples), although with **bootstrap.py**, one can inspect the convergence of the fitting parameters/errors to decide how many resamples are sufficient.

From the computed list of parameters, error measures can be computed. Assume that $\{x_i\}$ is the set of fitting parameters obtained. **bootstrap.py** and **make_fit.py** compute the arithmetic mean and standard deviation like this:

$$\bar{x}_{\text{arith}} = \frac{1}{N}\sum_i^N x_i, \tag{8.10}$$

$$\sigma_{\text{arith}}(x) = \sqrt{\frac{1}{N-1}\sum_i^N (x-\bar{x})^2}. \tag{8.11}$$

Because the distribution of $\{x_i\}$ might be skewed (e.g., contains some very large values but few very small ones), the script also computes the geometric mean and standard deviation like this:

$$\bar{x}_{\text{geom}} = \mathrm{e}^{\frac{1}{N}\sum_i^N \ln(x_i)}, \tag{8.12}$$

$$\sigma_{\text{geom}}(x) = \mathrm{e}^{\sqrt{\frac{1}{N-1}\sum_i^N (\ln(x)-\ln(\bar{x}))^2}} \tag{8.13}$$

Note that the geometric standard deviation is a dimensionless factor (unlike the arithmetic standard deviation, which has the same dimension as the mean). Therefore, within **bootstrap.py** and **make_fit.py** the geometric errors are always displayed with separate upper and lower bounds as $\bar{x}\,^{+\bar{x}(\sigma(x)-1)}_{-\bar{x}(1/\sigma(x)-1)}$. Note that **bootstrap.py** and **make_fit.py** will always report one times the standard deviation in the output. If larger confidence intervals are desired, simply multiply the arithmetic error as usual. For the geometric error, use for example $\bar{x}\,^{+\bar{x}(\sigma(x)^2-1)}_{-\bar{x}(1/\sigma(x)^2-1)}$.

8.5 Computing electronic populations

The electronic populations from SHARC trajectories can be obtained in different ways. This is primarily due to the fact that in surface hopping one could either consider the active state or the electronic wave function coefficients. This issue is discussed in detail in [?], where it is shown that the optimal way to obtain the electronic populations is actually a combination of both kinds of information in a Wigner-like transformation. Hence, there are three options in SHARC: (i) based on classical active state, (ii) based on electronic wave function coefficients, and (iii) based on Wigner-transform. Additionally, the populations can be analyzed in different representations (diagonal, MCH, diabatic).

Diagonal representation The diagonal representation, there is no basis transformation involved, and hence the equations are very simple. For a single trajectory, the three possible populations can be obtained by:

$$p_j^{\text{diag}(i)} = \delta_{j\alpha}, \tag{8.14}$$

$$p_j^{\text{diag}(ii)} = |c_j^{\text{diag}}|^2, \tag{8.15}$$

$$p_j^{\text{diag}(iii)} = \delta_{j\alpha}, \tag{8.16}$$

where α is the active diagonal state.

MCH representation To obtain the MCH representation, one needs to transform the populations with the matrix $\mathbf{U}$. Hence, one obtains:

$$p_j^{\text{MCH}(i)} = |U_{j\alpha}|^2, \tag{8.17}$$

$$p_j^{\text{MCH}(ii)} = |\sum_k U_{jk} c_k^{\text{diag}}|^2, \tag{8.18}$$

$$p_j^{\text{MCH}(iii)} = |U_{j\alpha}|^2 + \sum_{k<l} 2\Re(U_{jk}U_{jl}^* c_k^{\text{diag}} c_l^{\text{diag}*}). \tag{8.19}$$

Diabatic representation To obtain the diabatic representation, one needs to transform the populations with the appropriate transformation matrix **T**, which in SHARC is obtained as the matrix product of reference overlap matrix, time-ordered overlap matrices, and **U** for the current time step. Hence, one obtains:

$$p_j^{\mathrm{MCH}(i)} = |T_{j\alpha}|^2, \tag{8.20}$$

$$p_j^{\mathrm{MCH}(ii)} = |\sum_k T_{jk} c_k^{\mathrm{diag}}|^2, \tag{8.21}$$

$$p_j^{\mathrm{MCH}(iii)} = |T_{j\alpha}|^2 + \sum_{k<l} 2\Re(T_{jk} T_{jl}^* c_k^{\mathrm{diag}} c_l^{\mathrm{diag}*}). \tag{8.22}$$

8.6 Damping

If damping is activated in SHARC (keyword **`dampeddyn`**), in each time step the following modification to the velocity vector is made

$$\mathbf{v}' = \mathbf{v} \cdot \sqrt{C} \tag{8.23}$$

where C is the damping factor given in the input. Hence, in each time step the kinetic energy is modified by

$$E'_{\mathrm{kin}} = E_{\mathrm{kin}} \cdot C \tag{8.24}$$

The damping factor C must be between 0 and 1.

8.7 Decoherence

In surface hopping, without any corrections the coherence between the states is usually too large [?]. A trajectory in state β, but where state α has a large coefficient, will still travel according to the gradient of state β. However, the gradients of state α are almost certainly different to the ones of state β. As a consequence, too much population of state α is following the gradient of state β. Decoherence corrections damp in different ways the population of all states $\alpha \neq \beta$, so that only population of β follows the gradient of state β.

Currently, in SHARC there are two decoherence corrections implemented, "energy-based decoherence", or EDC, as given in [?] and "augmented fewest-switches surface hopping", or AFSSH, as described in [?].

8.7.1 Energy-based decoherence

In this scheme, after the surface hopping procedure, when the system is in state β, the coefficients are updated by the following relation

$$c'_\alpha = c_\alpha \cdot \exp\left[-\frac{1}{2}\Delta t \frac{|E_\alpha - E_\beta|}{\hbar}\left(1 + \frac{C}{E_{\mathrm{kin}}}\right)^{-1}\right], \qquad \alpha \neq \beta, \tag{8.25}$$

$$c'_\beta = \frac{c_\beta}{|c_\beta|} \cdot \left[1 - \sum_{\alpha\neq\beta} |c'_\alpha|^2\right]^{\frac{1}{2}} \tag{8.26}$$

where C is the decoherence parameter. The decoherence correction can be activated with the keyword **`decoherence`** in the input file. The decoherence parameter C can be set with the keyword **`decoherence_param`** (the default is 0.1 hartree, as suggested in [?]).

8.7.2 Augmented FSSH decoherence

Augmented FSSH by Subotnik and coworkers is described in [?]. For SHARC, the augmented FSSH algorithm was adjusted to the case of the diagonal representation.

The basic idea is that besides the actual trajectory, the program maintains an auxiliary trajectory for each state. The auxiliary trajectories are propagated using the gradients of the associated (not active) state, and because the gradients are different, the auxiliary trajectories eventually diverge from each other and from the main trajectory. From this diverging, one can compute decoherence rates which can be used to stochastically set the electronic coefficients of the diverging state to zero.

First, we compute two matrices:

$$\mathbf{S}^{\text{diag}}(t,t+\Delta t) = \mathbf{U}^\dagger(t)\mathbf{S}(t,t+\Delta t)\mathbf{U}(t+\Delta t), \tag{8.27}$$

$$\mathbf{H}^{\text{olddiag}}(t+\Delta t) = \mathbf{U}^\dagger(t)\mathbf{S}(t,t+\Delta t)\mathbf{H}^{\text{MCH}}(t+\Delta t)\mathbf{S}^\dagger(t,t+\Delta t)\mathbf{U}(t), \tag{8.28}$$

where S is the overlap matrix, as used in ??. Hence, $\mathbf{S}^{\text{diag}}(t,t+\Delta t)$ is the overlap matrix in the diagonal basis and $\mathbf{H}^{\text{olddiag}}(t+\Delta t)$ is the Hamiltonian at time $t+\Delta t$ expressed in the diagonal basis of time step t.

We then propagate the auxiliary trajectories for each state j, considering that the active state is α. For this, we need the gradient matrix $\mathbf{G}^{\text{diag}}$ from section ??. We define:

$$\sigma_j = |c_j(t)|^2. \tag{8.29}$$

We do the X step of the velocity-Verlet algorithm:

$$\mathbf{a}^j_A(t) = -\frac{((\mathbf{G}^{\text{diag}})_{jj} - (\mathbf{G}^{\text{diag}})_{\alpha\alpha})_A}{M_A}, \tag{8.30}$$

$$\mathbf{R}^j(t+\Delta t) = \mathbf{R}^j(t) + \mathbf{v}^j(t)\Delta t + \frac{1}{2}\mathbf{a}^j(t)\Delta t^2\sigma_j, \tag{8.31}$$

where A goes over the atoms. Then, we compute the new gradient:

$$\mathbf{g}^j(t+\Delta t) = -(\mathbf{G}^{\text{diag}})_{\alpha\alpha} + \sum_i (\mathbf{S}^{\text{diag}}(t,t+\Delta t))_{ji}(\mathbf{G}^{\text{diag}})_{ii}. \tag{8.32}$$

We then carry out the v step:

$$\mathbf{a}^j_A(t+\Delta t) = \frac{1}{2}\mathbf{a}^j_A(t) - \frac{(\mathbf{g}^j(t+\Delta t))_A}{M_A}, \tag{8.33}$$

$$\mathbf{v}^j(t+\Delta t) = \mathbf{v}^j(t) + \mathbf{a}^j_A(t+\Delta t)\Delta t\sigma_j. \tag{8.34}$$

We then perform a diabatization of the auxiliary trajectories (e.g., for a trivially avoided crossing, the moments between the crossing states are interchanged):

$$\mathbf{R}^j = \sum_i (\mathbf{S}^{\text{diag}}(t,t+\Delta t))_{ij}\mathbf{R}^i, \tag{8.35}$$

$$\mathbf{v}^j = \sum_i (\mathbf{S}^{\text{diag}}(t,t+\Delta t))_{ij}\mathbf{v}^i, \tag{8.36}$$

$$\tag{8.37}$$

to transform the data back into the adiabatic basis at time $t+\Delta t$.

Finally, we carry out the decoherence correction procedure for each auxiliary trajectory. We compute the displacement from the auxiliary trajectory of the active state:

$$\mathbf{D} = \mathbf{R}^j - \mathbf{R}^\alpha \tag{8.38}$$

We compute two rate constants:

$$r^j_1 = -\frac{1}{2}\mathbf{g}^{\mathbf{j}}(t+\Delta t)\cdot\mathbf{D}, \tag{8.39}$$

$$r^j_2 = 2|(\mathbf{G}^{\text{diag}})_{\alpha j}\cdot\mathbf{D}|, \tag{8.40}$$

or if no nonadiabatic coupling vectors are available:

$$r_2^j = 2\frac{|H_{\alpha j}^{\text{olddiag}}|}{\Delta t}\frac{\mathbf{D}\cdot\mathbf{v}}{\mathbf{v}\cdot\mathbf{v}}. \tag{8.41}$$

We draw a random number (identical for all states) r between 0 and 1. If $r < \Delta t(r_1^j - r_2^j)$, then we collapse state j, by setting its coefficient to zero and enlarging the coefficient of the active state such that the total norm is conserved; we also set the moments $\mathbf{R}^j$ and $\mathbf{v}^j$ to zero. If no collapse occurred, if $r < -\Delta t r_1^j$, we set the moments $\mathbf{R}^j$ and $\mathbf{v}^j$ to zero.

After a surface hop occurred, we reset all auxiliary trajectories.

8.8 Essential Dynamics Analysis

As an alternative to normal mode analysis (see section **??**), essential dynamics analysis can be used to identify important modes in the dynamics. This procedure is a principal component analysis of the geometric displacements.[**? ?**] Unlike normal mode analysis, it does not depend on the availability of the normal mode vectors.

The covariance matrix is computed from the following equation (μ and ν are indices over the $3N_{\text{atom}}$ degrees of freedom):

$$\bar{R}_\mu = \frac{1}{N_{\text{traj}}(k_{\text{end}} - k_{\text{start}})}\sum_{i=1}^{N_{\text{traj}}}\sum_{k=k_{\text{start}}}^{k_{\text{end}}} R_\mu^i(k\Delta t) \tag{8.42}$$

and

$$A_{\mu\nu} = \frac{1}{N_{\text{traj}}(k_{\text{end}} - k_{\text{start}})}\sum_{i=1}^{N_{\text{traj}}}\sum_{k=k_{\text{start}}}^{k_{\text{end}}} R_\mu^i(k\Delta t)R_\nu^i(k\Delta t) \tag{8.43}$$

as a matrix **C** with elements:

$$C_{\mu\nu} = A_{\mu\nu} - R_\mu R_\nu. \tag{8.44}$$

Diagonalization of the symmetric matrix **C** gives a set of eigenvalues and eigenvectors. The eigenvectors represent the essential dynamics modes, and the corresponding eigenvalues are the variances of the modes. Modes with large variances show strong motion in the dynamics, whereas small variances are found for modes which show weak motion. Because the modes are uncorrelated, the few modes with the largest variance describe most of the molecular motion, which shows that essential dynamics analysis can be used to for data reduction.

8.9 Excitation Selection

`excite.py` can select initial active states for the dynamics based on the excitation energies $E_{k,\alpha}$ and the oscillator strengths $f_{k,\alpha}^{\text{osc}}$ for each initial condition k and excited state α.

First, for all excited states of all initial conditions, the maximum value $p_{\max}$ of

$$p_{k,\alpha} = \frac{f_{k,\alpha}^{\text{osc}}}{E_{k,\alpha}^2} \tag{8.45}$$

is found. Then for each excited state, a random number $r_{k,\alpha}$ is picked from $[0, 1]$. If

$$r_{k,\alpha} < \frac{p_{k,\alpha}}{p_{\max}} \tag{8.46}$$

then the excited state is selected as a valid initial condition. This excited-state selection scheme is taken from [**?**].

Within **`excite.py`** it is possible to restrict the selection to a subset of all excited states (only certain adiabatic states/within a given energy range). In this case, also $p_{\max}$ is only determined based on this subset of excited states.

8.9.1 Excitation Selection with Diabatization

The active states can be selected based on a diabatization. This necessitates the wave function overlaps between a reference geometry (**ICOND_00000/**) and the current initial condition k.

The overlap matrix with elements

$$S_{ij}^{0k} = \left\langle \Psi_i(\mathbf{R_0}) | \Psi_j(\mathbf{R}_k) \right\rangle \tag{8.47}$$

can be computed with **wfoverlap.x** (calculations can be setup with **setup_init.py**). This overlap matrix is rescaled during the excitation selection procedure such S_{11}^{0k} is equal to one (by dividing all elements by $|S_{11}^{0k}|^2$). Then, assume we want to start all trajectories in the state which corresponds to state x at the reference geometry. The excitation selection will select a state y as initial state if $S_{xy}^{0k} > 0.5$.

8.10 Global fits and kinetic models

In this section, we specify the basic assumptions of the chemical kinetic models used with the scripts **make_fitscript.py** and **make_fit.py** and the globals fits of these models to data.

8.10.1 Reaction networks

The kinetic models used by **make_fitscript.py** and **make_fit.py** are based on a chemical reaction network, where chemical *species* react via unimolecular *reactions*.

The reaction networks allowed in the script are a simple directed graphs, where the species are the vertices and the reactions are the directed edges. Each reaction is characterized by an associated rate constant and connects exactly one reactant species to exactly one product species.

In order to obtain rate laws which can be integrated easily, there are a number of restrictions imposed on the network graphs. Some restrictions and possible features of the graphs are depicted exemplarily in Figure **??**. First, each species and each reaction rate needs to have a unique label. There cannot be two species with the same label, but there can be two reactions with the same reaction rate constant. Second, the graph must be a simple directed graph, hence there cannot be any (self-) loops or more than one reaction with the same initial and the same final species. All restrictions marked as "Not allowed" in the Figure are enforced in the input dialogue of **make_fitscript.py** However, back reactions are allowed (as a back reaction has a different initial and a different final species as the corresponding reaction). Except from these restrictions, the graph may contain combinations of sequential and parallel reactions. The graph may also be disjoint, i.e., it can be the union of several independent sub-networks. Disjoint graphs with repeated reaction labels can be useful to fit population data from ensembles with identical settings but different initial conditions (in this case, merge the population files with **paste** before starting the fit.

There are two kinds of cycles possible, called here *parallel pathways* and *closed walks*. Parallel pathways are independent sequences of reactions with the same initial and final species (e.g., $A \longrightarrow C$ and $A \longrightarrow B \longrightarrow C$). A closed walk is a sequence of reactions where the initial species is equal to the final species. These cycles can sometimes lead to problems. If you use **make_fitscript.py**, then it is necessary that the system of differential equations of the rate laws can be integrated in closed form by MAXIMA. Since **make_fit.py** solves the differential equation system numerically, it is not necessary that the solution can be given in closed form. However, cycles can also lead to problems in the fitting procedure (rate constants in parallel pathways can be strongly correlated and cause large errors and bad convergence in fitting).

An example reaction network graph is shown in Figure **??**. In this graph, there are 5 species (S_2, S_1, S_0, T_2, and T_1) and 6 reactions, each with a rate constant (k_S, k_{Rlx}, k_{22}, k_{11}, k_{21}, k_{12}). This graph shows some features which are allowed in the reaction networks for **make_fitscript.py**: sequential reactions, parallel reactions, back reactions, and converging reaction pathways. Note that this reaction network is likely to cause problems in the fitting step (large errors).

8.10.2 Kinetic models

Based on the reaction network graph, a system of differential equations describing the rate laws of all species can be setup. The system of equations (equivalently, the matrix differential equation) can be written as:

$$\frac{\partial}{\partial t}\mathbf{s}(t) = \mathbf{A} \cdot \mathbf{s}(t), \tag{8.48}$$

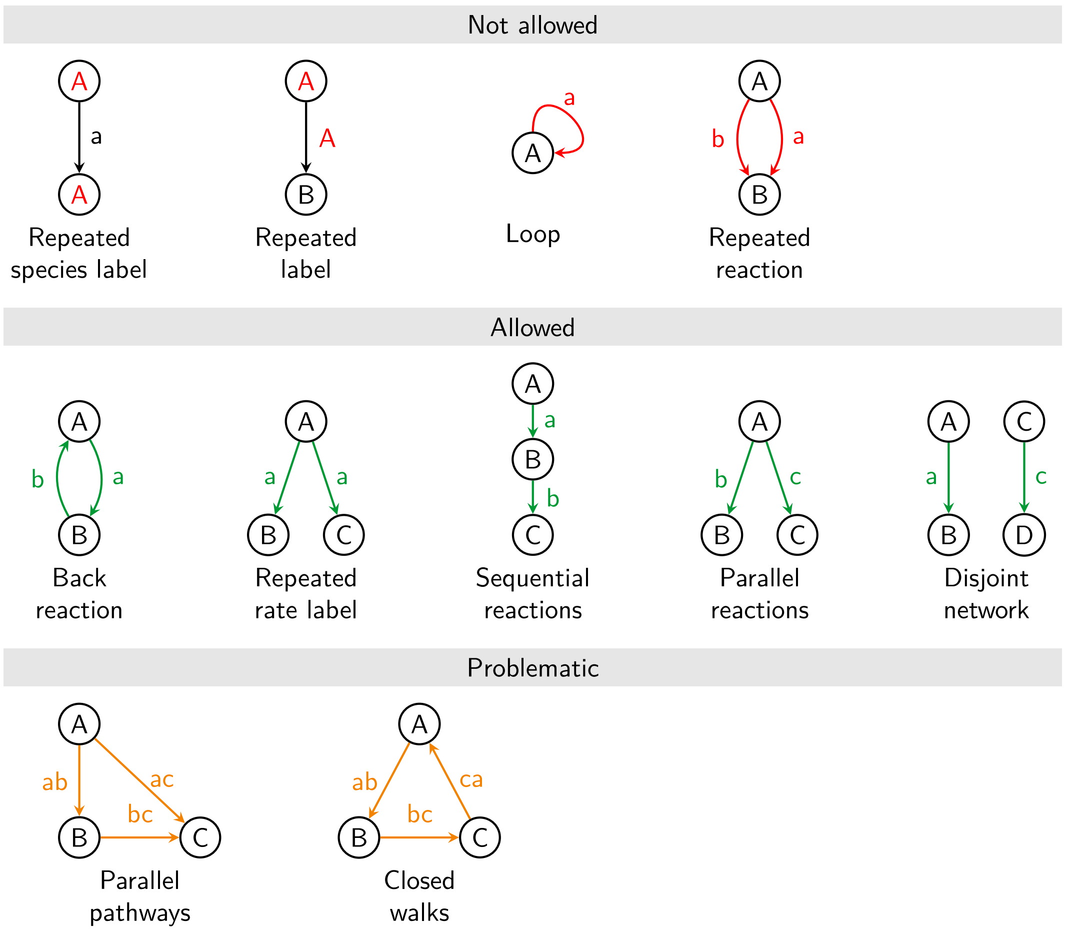

Figure 8.1: Forbidden and allowed features of the reaction network graphs.

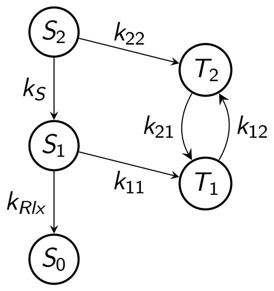

Figure 8.2: Example reaction network graph. For explanation see text.

where **s** is the vector of the time-dependent populations of each species and **A** is the matrix containing the rate constants. In order to construct **A**, start with $\mathbf{A} = 0$ and, for each reaction from species i to species j with rate k, substract k from A_{ii} and add k to A_{ji}.

In order to integrate this system of equations, in practice one also needs to define initial conditions. In the present context, the initial conditions are fully specified by $\mathbf{s}(t = 0) = \mathbf{s}_0$, where $\mathbf{s}_0$ are constant expressions defined by the user.

Solving (??) yields (in not too complicated cases) the closed-form expressions for the functions $\mathbf{s}(t)$, which contain as parameters all rate constants k and all initial values $\mathbf{s}_0$.

8.10.3 Global fit

Suppose there is a data set $\mathbf{p}(t) = (p_1(t), p_2(t), \dots)$ of time-dependent populations of several states $k = 1, 2, \dots$ and which is given at several points of time $\{t_i : 0 < t_i < t_{\max}\}$. We can *fit* to one data set $p_k(t)$ a function $s_l(t)$ from $\mathbf{s}(t)$ by optimizing the parameters (mainly the rate constants) such that $\sum_i |p_k(t_i) - s_l(t_i)|^2$ becomes minimal.

In order to perform global fit including several species l and several states k, we construct piecewise global functions from $\mathbf{p}(t)$ and $\mathbf{s}(t)$ and then optimize the parameters accordingly.

8.11 Gradient transformation

Since the actual dynamics is performed on the PESs of the diagonal states, also the gradients have to be transformed to the diagonal representation. To this end, first a generalized gradient matrix $\mathbf{G}^{\mathrm{MCH}}$ is constructed from the gradients $\mathbf{g}_\alpha^{\mathrm{MCH}} = -\nabla_{\mathbf{R}} H_{\alpha\alpha}^{\mathrm{MCH}}$ and the nonadiabatic coupling vectors $\mathbf{K}_{\beta\alpha}^{\mathrm{MCH}}$:

$$\mathbf{G}^{\mathrm{MCH}} = \begin{pmatrix} \mathbf{g}_1 & -(H_{11}-H_{22})\mathbf{K}_{12} & -(H_{11}-H_{33})\mathbf{K}_{13} & \cdots \\ -(H_{22}-H_{11})\mathbf{K}_{21} & \mathbf{g}_2 & -(H_{22}-H_{33})\mathbf{K}_{23} & \cdots \\ -(H_{33}-H_{11})\mathbf{K}_{31} & -(H_{33}-H_{22})\mathbf{K}_{32} & \mathbf{g}_3 & \\ \vdots & \vdots & & \ddots \end{pmatrix} \tag{8.49}$$

Note that all quantities in the matrix are in the MCH representation, the superscripts were omitted for brevity.

The matrix $\mathbf{G}^{\mathrm{MCH}}$ is subsequently transformed into the diagonal representation, using the transformation matrix $\mathbf{U}$:

$$\mathbf{G}^{\mathrm{diag}} = \mathbf{U}^\dagger \mathbf{G}^{\mathrm{MCH}} \mathbf{U}. \tag{8.50}$$

The diagonal elements of $\mathbf{G}^{\mathrm{diag}}$ now contain the gradients of the diagonal states, while the off-diagonal elements contain the scaled nonadiabatic couplings $-(H_{\beta\beta}^{\mathrm{diag}} - H_{\alpha\alpha}^{\mathrm{diag}})\mathbf{K}_{\beta\alpha}^{\mathrm{diag}}$. The gradients are needed in the Velocity Verlet algorithm in order to propagate the nuclear coordinates. The nonadiabatic couplings are necessary if the velocity vector is to be rescaled along the nonadiabatic coupling vector.

Since the matrix $\mathbf{G}^{\mathrm{MCH}}$ contains elements which are itself vectors with $3N$ components, the transformation is done component-wise (e.g. first a matrix $G_{x,\mathrm{atom1}}$ is constructed from the x components of all gradients (and nonadiabatic couplings) for atom 1, this matrix is transformed and written to the x, atom 1 component of $\mathbf{G}^{\mathrm{diag}}$, then this is repeated for all components).

Since the calculation of the nonadiabatic couplings $\mathbf{K}_{\beta\alpha}^{\mathrm{MCH}}$ might add considerable computational cost, there is the input keyword **`nogradcorrect`** which tells SHARC to neglect the $\mathbf{K}_{\beta\alpha}^{\mathrm{MCH}}$ in the gradient transformation.

8.11.1 Dipole moment derivatives

For strong laser fields, the derivative of the dipole moments might be a non-negligible contribution to the gradients. In this case, they should be included in the gradient transformation step:

$$\mathbf{G}^{\mathrm{diag}} = \mathbf{U}^\dagger \left[\mathbf{G}^{\mathrm{MCH}} - \boldsymbol{\epsilon}(t) \cdot \frac{\partial}{\partial R} \boldsymbol{\mu} \right] \mathbf{U}. \tag{8.51}$$

This can be activated with the keyword **`dipole_grad`** in the SHARC input file.

8.12 Internal coordinates definitions

In this section, the internal coordinates available in **`geo.py`** are defined.

Bond length The bond length between two atoms a and b is:

$$r_{ab} = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2 + (z_a - z_b)^2} \tag{8.52}$$

Bond angle The bond angle for atoms a, b and c is defined as the angle

$$\theta = \cos^{-1}\left(\frac{\mathbf{v}_{ba} \cdot \mathbf{v}_{bc}}{|\mathbf{v}_{ba}| \cdot |\mathbf{v}_{bc}|}\right) \tag{8.53}$$

where $\mathbf{v}_{ba}$ is the vector from atom b to atom a.

Dihedral The dihedral angle is defined via the vectors $\mathbf{w}_1$ and $\mathbf{w}_2$:

$$\mathbf{w}_1 = \mathbf{v}_{ab} \times \mathbf{v}_{bc} \qquad \text{and} \qquad \mathbf{w}_2 = \mathbf{v}_{bc} \times \mathbf{v}_{cd}. \tag{8.54}$$

The dihedral is given as the angle between $\mathbf{w}_1$ and $\mathbf{w}_2$ according to equation (**??**). In order to distinguish left- and right-handed rotation, also the vector $\mathbf{Q} = \mathbf{w}_1 \times \mathbf{w}_2$ is computed, and if the angle between $\mathbf{Q}$ and $\mathbf{v}_{bc}$ is larger than 90° then the value of the dihedral is multiplied by -1.

Pyramidalization angle The pyramidalization angle is defined via the vectors $\mathbf{v}_{ba}$ and

$$\mathbf{w}_1 = \mathbf{v}_{bc} \times \mathbf{v}_{bd}. \tag{8.55}$$

The pyramidalization angle is then given as $90° - \theta(\mathbf{v}_{ba}, \mathbf{w}_1)$. This definition of the pyramidalization angle works best for nearly planar arrangements, like in amino groups.

An alternative definition of the pyramidalization angle works better for strongly pyramidalized situations (e.g., *fac*-arranged atoms in a octahedral metal complex). This pyramidalization angle is defined as 180° minus the angle between $\mathbf{v}_{ab}$ and the average of $\mathbf{v}_{bc}$ and $\mathbf{v}_{bd}$.

Cremer-Pople parameters The definitions of the Cremer-Pople parameters for 5- and 6-membered rings is described in [**?**].

Boeyens classification For 6-membered rings, the Boeyens classification scheme is described in [**?**].

Angle between two rings In order to compute angles between the mean planes of two rings, we compute the normal vector of the two rings as in [**?**]. We then compute the angle between the two normal vectors.

8.13 Kinetic energy adjustments

There are several options how to adjust the kinetic energy after a surface hop occurred. The simplest option is to not adjust the kinetic energy at all (input: **`ekincorrect none`**), but this obviously leads to violation of the conservation of total energy.

Alternatively, the velocities of all atoms can be rescaled so that the new kinetic energy and the potential energy of the new state β again sum up to the total energy.

$$f = \sqrt{\frac{E_{\text{total}} - E_\beta}{E_{\text{kin}}}} \tag{8.56}$$

$$\mathbf{v}' = f\mathbf{v} \tag{8.57}$$

$$E'_{\text{kin}} = f^2 E_{\text{kin}} \tag{8.58}$$

Within this methodology, when the energy of the old state α was lower than the energy of the new state β the kinetic energy is lowered. Since the kinetic energy must be positive, this implies that there might be states which cannot be reached (their potential energy is above the total energy). A hop to such a state is called "frustrated hop" and will be rejected by SHARC. Rescaling parallel to the nuclear velocity vector is requested with **`ekincorrect parallel_vel`**.

Alternatively, according to Tully's original formulation of the surface hopping method [?], after a hop from α to β only the component of the velocity along the direction of the nonadiabatic coupling vector $\mathbf{K}_{\beta\alpha}$ should be rescaled. With

$$a = \sum_{i}^{N_{\text{atom}}} \frac{\mathbf{K}_{\beta\alpha,i} \cdot \mathbf{K}_{\beta\alpha,i}}{2M_i} \tag{8.59}$$

$$b = \sum_{i}^{N_{\text{atom}}} \mathbf{v}_i \cdot \mathbf{K}_{\beta\alpha,i} \tag{8.60}$$

the available energy can be calculated:

$$\Delta = 4a\left(E_\alpha - E_\beta\right) + b^2. \tag{8.61}$$

If $\Delta < 0$, the hop is frustrated and will be rejected. Otherwise, the scaled velocities $\mathbf{v}'$ can be calculated as

$$\mathbf{v}'_i = \mathbf{v}_i - f\frac{\mathbf{K}_{\beta\alpha,i}}{M_i} \tag{8.62}$$

$$\text{with} \quad f = \begin{cases} \frac{b+\sqrt{\Delta}}{2a}, & b < 0, \\ \frac{b-\sqrt{\Delta}}{2a}, & b \geq 0. \end{cases} \tag{8.63}$$

This procedure can be requested with **`ekincorrect parallel_nac`**. Note that in this case SHARC will request the nonadiabatic coupling vectors, even if they are not used for the wave function propagation.

8.13.1 Reflection for frustrated hops

As suggested by Tully [?], during a frustrated hop one might want to reflect the trajectory (i.e., invert some component of the velocity vector). In SHARC, there are three possible options to this, the first being no reflection (**`reflect_frustrated none`**). Alternatively, one can invert the total velocity vector $\mathbf{v} := -\mathbf{v}$ (**`reflect_frustrated parallel_vel`**).

As this leads to a nearly complete time reversal and might be inappropriate, as a third option one can choose to only reflect the velocity component parallel to the nonadiabatic coupling vector between the active state and the state to which the frustrated hop was attempted. The condition for reflection in this case is based on three scalar products:

$$k_1 = \mathbf{g}_\alpha \cdot \mathbf{t}_{\alpha f}, \tag{8.64}$$

$$k_2 = \mathbf{g}_f \cdot \mathbf{t}_{\alpha f}, \tag{8.65}$$

$$k_3 = \sum_A M_A v_A (t_{\alpha f})_A, \tag{8.66}$$

Reflection is only carried out if $k_1 k_2 < 0$ and $k_2 k_3 < 0$. In order to reflect, we compute:

$$\mathbf{v}_A := \mathbf{v}_A - 2\frac{\mathbf{v}_A \cdot \mathbf{t}_A}{\mathbf{t}_A \cdot \mathbf{t}_A}\mathbf{t}_A. \tag{8.67}$$

where $\mathbf{t}_A$ is the component of $\mathbf{t}_{\alpha f}$ corresponding to atom A.

8.14 Laser fields

The program **`laser.x`** can calculate laser fields as superpositions of several analytical, possibly chirped, laser pulses. In the following, the laser parametrization is given (see [?] for further details).

8.14.1 Form of the laser field

In general, the laser field $\boldsymbol{\epsilon}(t)$ is a linear superposition of a number of laser pulses $l_i(t)$:

$$\boldsymbol{\epsilon}(t) = \sum_i \mathbf{p}_i l_i(t), \tag{8.68}$$

where $\mathbf{p}_i$ is the normalized polarization vector of pulse i.

A pulse $l(t)$ is formed as the product of an envelope function and a field function.

$$l(t) = \mathcal{E}(t)f(t) \tag{8.69}$$

8.14.2 Envelope functions

There are two types of envelope function defined in **`laser.x`**, which are Gaussian and sinusoidal.

The Gaussian envelope is defined as:

$$\mathcal{E}(t) = \mathcal{E}_0 \mathrm{e}^{-\beta(t-t_c)^2} \tag{8.70}$$

$$\beta = \frac{4\ln 2}{\mathrm{FWHM}^2} \tag{8.71}$$

where $\mathcal{E}_0$ is the peak field strength, FWHM is the full width at half maximum and t_c is the temporal center of the pulse.

The sinusoidal envelope is defined as:

$$\mathcal{E}(t) = \mathcal{E}_0 \begin{cases} 0 & \text{if } t < t_0, \\ \sin^2\left(\frac{\pi(t-t_0)}{2(t_c-t_0)}\right) & \text{if } t_0 < t < t_c, \\ 1 & \text{if } t_c < t < t_{c2}, \\ \cos^2\left(\frac{\pi(t-t_{c2})}{2(t_e-t_{c2})}\right) & \text{if } t_{c2} < t < t_e, \\ 0 & \text{if } t_e < t, \end{cases} \tag{8.72}$$

where again $\mathcal{E}_0$ is the peak field strength, t_0 and t_c define the interval where the field strength increases, and t_{c2} and t_e define the interval where the field strength decreases. Figure **??** shows the general form of the envelope functions and the meaning of the temporal parameters t_0, t_c, t_{2c} and t_e.

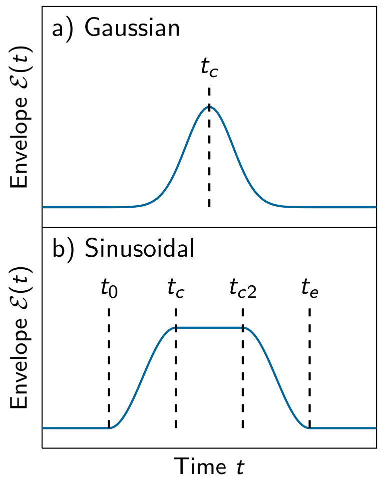

Figure 8.3: Types of laser envelopes implemented in **`laser.x`**.

8.14.3 Field functions

The field function $f(t)$ is defined as:

$$f(t) = \mathrm{e}^{\mathrm{i}(\omega_0(t-t_c)+\phi)}, \tag{8.73}$$

where ω_0 is the central frequency and ϕ is the phase of the pulse. Even though the laser field is complex in this expression, in the propagation of the electronic wave function in SHARC only the real part is used.

8.14.4 Chirped pulses

In order to apply a chirp to the laser pulse $l(t)$, it is first Fourier transformed to the frequency domain, giving the function $\tilde{l}(\omega)$. The chirp is applied by calculating:

$$\tilde{l}'(\omega) = \tilde{l}(\omega)\mathrm{e}^{-i\left[b_1|\omega-\omega_0|+\frac{b_2}{2}(\omega-\omega_0)^2+\frac{b_3}{6}(\omega-\omega_0)^3+\frac{b_4}{24}(\omega-\omega_0)^4\right]} \tag{8.74}$$

The chirped laser in the time domain $l'(t)$ is then obtained by Fourier transform of the chirped pulse $\tilde{l}'(\omega)$.

8.14.5 Quadratic chirp without Fourier transform

If **`laser.x`** was compiled without the FFTW package, the only accessible chirps are quadratic chirps for Gaussian pulses:

$$l(t) = \mathcal{E}_0'\mathrm{e}^{-\beta'(t-t_c)^2}\mathrm{e}^{-\mathrm{i}\left(\omega_0(t-t_c)+a_2(t-t_c)^2+\phi\right)} \tag{8.75}$$

$$\beta = \frac{4\ln 2}{\mathrm{FWHM}^2} \tag{8.76}$$

$$\beta' = \frac{1}{\frac{1}{\beta}+4\beta b_2^2} \tag{8.77}$$

$$a_2 = \frac{b_2}{\frac{1}{4\beta^2}+b_2^2} \tag{8.78}$$

$$\mathcal{E}_0' = \mathcal{E}_0\sqrt{\frac{1}{2\mathrm{i}b_2\beta+1}} \tag{8.79}$$

Other chirps are only possible with the Fourier transformation.

8.15 Laser interactions

The laser field $\boldsymbol{\epsilon}$ is included in the propagation of the electronic wave function. In each substep of the propagation, the interaction of the laser field with the dipole moments is included in the Hamiltonian. The contribution $\mathbf{V}_i$ is in each time step added to the Hamiltonian in equations (**??**) or (**??**), respectively:

$$\mathbf{V}_i = -\Re\left(\boldsymbol{\mu}_i\cdot\boldsymbol{\epsilon}_i\right), \tag{8.80}$$

$$\boldsymbol{\mu}_i = \boldsymbol{\mu}^{\mathrm{MCH}}(t) + \frac{i}{n}\left(\boldsymbol{\mu}^{\mathrm{MCH}}(t+\Delta t) - \boldsymbol{\mu}^{\mathrm{MCH}}(t)\right), \tag{8.81}$$

$$\boldsymbol{\epsilon}_i = \boldsymbol{\epsilon}\left(t+\frac{i}{n}\Delta t\right) \tag{8.82}$$

where i, n t and Δt are defined as in section **??**.

8.15.1 Surface Hopping with laser fields

If laser fields are present, there can be two fundamentally different types of hops: laser-induced hops and nonadiabatic hops. The latter ones are the same hops as in the laser-free simulations, and demand that the total energy is conserved. The laser-induced hops on the other hand demand that the momentum (kinetic energy) is conserved. Hence, SHARC needs to decide for every hop whether it is laser-induced or not.

Consider a previous state α and a new state β. Currently, the hop is classified based on the energy gap $\Delta E = |E_\beta^{\mathrm{diag}} - E_\alpha^{\mathrm{diag}}|$ and the instantaneous central energy of the laser pulse ω. The hop is assumed to be laser-induced if

$$|\Delta E - \omega| < W, \tag{8.83}$$

where W is a fixed parameter. W can be set using the input keyword **`laserwidth`**.

If a hop has been classified as laser-free, the momentum is adjusted according to the equations given in section **??**.

8.16 Linear Vibronic Coupling Models

In the vibronic coupling model [?] the diabatic energy and coupling matrix **V** is constructed as

$$\mathbf{V} = V_0\mathbf{1} + \mathbf{W}, \tag{8.84}$$

where V_0 is the reference potential and **W** includes the state-specific vibronic terms.

Within SHARC, the reference potential is chosen to be harmonic. The reference potential is expressed in dimensionless mass-frequency-scaled normal coordinates, which can be computed from the Cartesian coordinates r_A as

$$Q_i = \sqrt{\omega_i}\sum_A K_{Ai}\sqrt{M_A}\left(r_A - r_A^{\mathrm{ref}}\right) \tag{8.85}$$

where ω_i is the frequency of normal mode i, M_A is an atomic mass, and K_{Ai} denotes the conversion matrix between mass-weighted Cartesian and normal coordinates. Using these coordinates, the harmonic reference potential is given as

$$V_0 = \sum_i \frac{\omega_i}{2}Q_i^2. \tag{8.86}$$

In the case of the linear vibronic coupling model, one additionally considers the following state-specific terms that constitute the **W** matrix.

$$W_{\alpha\alpha} = \epsilon_\alpha + \sum_i \kappa_i^{(\alpha)}Q_i, \tag{8.87}$$

$$W_{\alpha\beta} = \eta_{\alpha\beta} + \sum_i \lambda_i^{(\alpha\beta)}Q_i, \quad \alpha \neq \beta \tag{8.88}$$

Here the ϵ_α are the vertical excitation energies, the $\eta_{\alpha\beta}$ are the SOC constants, and the $\kappa_i^{(\alpha)}$ and $\lambda_i^{(\alpha\beta)}$ are termed intrastate and interstate vibronic coupling constant [?].

8.16.1 Obtaining LVC parameters from ab initio data

All LVC parameters are either constant or linear, implying that they can be obtained from either a single ab initio computation or from some first derivative. The following equations show how the parameters ϵ_α, $\eta_{\alpha\beta}$, $\kappa_i^{(\alpha)}$, and $\lambda_i^{(\alpha\beta)}$ are obtained.

The parameters ϵ_α are simply the vertical excitation energies at the reference geometry:

$$\epsilon_\alpha = H_{\alpha\alpha}^{\mathrm{MCH}}(\vec{Q}=0) = H_{\alpha\alpha}^{\mathrm{MCH}}(\vec{r}^{\mathrm{ref}}) - E^{\mathrm{ref}}, \tag{8.89}$$

where E^{ref} is some reference energy. Likewise, the SOC parameters $\eta_{\alpha\beta}$ are simply the SOC matrix elements at the reference geometry:

$$\eta_{\alpha\beta} = H_{\alpha\beta}^{\mathrm{MCH}}(\vec{Q}=0) = H_{\alpha\beta}^{\mathrm{MCH}}(\vec{r}^{\mathrm{ref}}). \tag{8.90}$$

These two equations hold because in the LVC model we assume that at the reference geometry diabatic basis and MCH basis coincide.

The intrastate linear vibronic coupling term $\kappa_i^{(\alpha)}$ is the gradient of the diabatic energy of state n. Since at the reference geometry, diabatic and MCH basis coincide, this is equivalent to the gradient of the corresponding MCH state. This gradient needs only be transformed from Cartesian coordinates into normal mode coordinates:

$$\kappa_i^{(\alpha)} = \frac{1}{\sqrt{\omega_i}}\sum_A \frac{K_{Ai}}{\sqrt{M_A}}\left.\frac{\partial H_{\alpha\alpha}^{\mathrm{MCH}}}{\partial r_A}\right|_{\vec{r}=\vec{r}^{\mathrm{ref}}} \tag{8.91}$$

The interstate linear vibronic coupling term $\lambda_i^{(\alpha\beta)}$ is the gradient of the diabatic coupling between states n and m. If analytical nonadiabatic coupling vectors are available, these parameters can be obtained—similarly as the $\kappa_i^{(\alpha)}$ parameters—by coordinate transformation of the energy-difference-scaled nonadiabatic coupling vector:

$$\lambda_i^{(\alpha\beta)} = \frac{1}{\sqrt{\omega_i}}\sum_A \frac{K_{Ai}}{\sqrt{M_A}}\left(H_{\alpha\alpha}^{\mathrm{MCH}}(\vec{r}^{\mathrm{ref}}) - H_{\beta\beta}^{\mathrm{MCH}}(\vec{r}^{\mathrm{ref}})\right)\left.\left\langle\psi_n\middle|\frac{\partial}{\partial r_A}\middle|\psi_m\right\rangle\right|_{\vec{r}=\vec{r}^{\mathrm{ref}}} \tag{8.92}$$

If gradients or nonadiabatic coupling vectors are not available, then the $\kappa_i^{(\alpha)}$ and $\lambda_i^{(\alpha\beta)}$ parameters can be obtained numerically. To this end, one performs ab initio calculations at displaced geometries $\pm\delta Q_i$, where

$$r_A^{\pm i} = r_A^{\text{ref}} \pm \frac{\delta Q_i K_{Ai}}{\sqrt{M_A \omega_i}}. \tag{8.93}$$

The ab initio calculations at these geometries provide the energies $H_{\alpha\alpha}^{\text{MCH}}(r_A^{\pm i})$ and the wave function overlaps $S_{\alpha\beta}^{\pm i} = \langle \psi_\alpha^{\text{MCH}}(\vec{r}^{\text{ref}}) | \psi_\beta^{\text{MCH}}(\vec{r}^{\pm i}) \rangle$. From these data, the $\kappa_i^{(\alpha)}$ values can be computed as:

$$\kappa_i^{(\alpha)} = \frac{1}{2\delta Q_i} \left((\mathbf{S}^{+i}\mathbf{H}^{\text{MCH}}(\vec{r}^{+i})(\mathbf{S}^{+i})^T)_{\alpha\alpha} - (\mathbf{S}^{-i}\mathbf{H}^{\text{MCH}}(\vec{r}^{-i})(\mathbf{S}^{-i})^T)_{\alpha\alpha} \right), \tag{8.94}$$

and the $\lambda_i^{(\alpha\beta)}$ as:

$$\lambda_i^{(\alpha\beta)} = \frac{1}{2\delta Q_i} \left((\mathbf{S}^{+i}\mathbf{H}^{\text{MCH}}(\vec{r}^{+i})(\mathbf{S}^{+i})^T)_{\alpha\beta} - (\mathbf{S}^{-i}\mathbf{H}^{\text{MCH}}(\vec{r}^{-i})(\mathbf{S}^{-i})^T)_{\alpha\beta} \right). \tag{8.95}$$

8.17 Normal Mode Analysis

The normal mode analysis can be used to find important vibrational modes in the excited-state dynamics.[? ?]
Given a matrix **Q** containing the normal mode vectors and a reference geometry $\mathbf{R}^{\text{ref}}$, calculate for each trajectory

$$R^i(t) = \mathbf{Q}^{-1}(\mathbf{R}^i(t) - \mathbf{R}^{\text{ref}}) \tag{8.96}$$

to obtain the displacements in normal mode coordinates. Averaging over the displacements gives the average trajectory:

$$\bar{R}(t) = \frac{1}{N_{\text{traj}}} \sum_{i=1}^{N_{\text{traj}}} R^i(t) \tag{8.97}$$

which should contain only coherent motion, since random motion cancels out in an ensemble.
A measure for the coherent activity in a mode is the standard deviation (over time) of the average trajectory:

$$R_{\text{coh}}^2 = \frac{1}{k_{\text{end}} - k_{\text{start}}} \sum_{k=k_{\text{start}}}^{k_{\text{end}}} \bar{R}(k\Delta t)^2 - \left(\frac{1}{k_{\text{end}} - k_{\text{start}}} \sum_{k=k_{\text{start}}}^{k_{\text{end}}} \bar{R}(k\Delta t) \right)^2, \tag{8.98}$$

where k_{start} and k_{end} are the start and end time steps for the analysis. R_{coh} a vector with one number per normal mode, where larger number mean that there is more coherent activity in this mode.
A measure for the total motion in a mode is the total standard deviation:

$$R_{\text{total}}^2 = \frac{1}{N_{\text{traj}}(k_{\text{end}} - k_{\text{start}})} \sum_{i=1}^{N_{\text{traj}}} \sum_{k=k_{\text{start}}}^{k_{\text{end}}} R^i(k\Delta t)^2 - \left(\frac{1}{N_{\text{traj}}(k_{\text{end}} - k_{\text{start}})} \sum_{i=1}^{N_{\text{traj}}} \sum_{k=k_{\text{start}}}^{k_{\text{end}}} R^i(k\Delta t) \right)^2. \tag{8.99}$$

8.18 Optimization of Crossing Points

With `orca_External` it is possible to optimize different kinds of crossing points. In all cases, these optimizations involve the energies of the lower state E_l and upper state E_u, the energy difference $\Delta E = E_u - E_l$, the gradients of the lower state $\mathbf{g}_l$ and upper state $\mathbf{g}_u$, the gradient difference vector **d**, and/or the nonadiabatic coupling vector **t**.

The simplest case is the optimization of minimum-energy crossing points between states of different multiplicity, because in this case the nonadiabatic coupling vector is zero and the branching space is one-dimensional. In this case [?

], the energy to optimize is E_u inside the intersection space, and ΔE inside the branching space. The corresponding gradient to follow $\mathbf{F}$ can be written as:

$$\mathbf{F} = \mathbf{g}_u - \frac{\mathbf{g}_u \cdot \mathbf{d}}{\mathbf{d} \cdot \mathbf{d}}\mathbf{d} + 2(E_u - E_l)\frac{\mathbf{d}}{|\mathbf{d}|}. \tag{8.100}$$

More complicated is the optimization of a conical intersection, between states of the same multiplicity, because the branching space is two-dimensional. The corresponding gradient to follow $\mathbf{F}$ is:

$$\mathbf{F} = \mathbf{g}_u - \frac{\mathbf{g}_u \cdot \mathbf{d}}{\mathbf{d} \cdot \mathbf{d}}\mathbf{d} - \frac{\mathbf{g}_u \cdot \mathbf{t}}{\mathbf{t} \cdot \mathbf{t}}\mathbf{t} + 2(E_u - E_l)\frac{\mathbf{d}}{|\mathbf{d}|}, \tag{8.101}$$

where $\mathbf{d}$ and $\mathbf{t}$ need to be orthogonalized.

If no nonadiabatic coupling vector is available because the interface cannot deliver them, conical intersections are optimized with the penalty function method of Levine et al. [**?**]. The effective energy to optimize is defined as:

$$E_{\text{eff}} = \frac{E_l + E_u}{2} + \sigma\frac{(E_u - E_l)^2}{E_u - E_l + \alpha}. \tag{8.102}$$

This equation is a combination of the two main targets of the optimization, the average energy and the energy gap. The parameter σ allows prioritizing either of the two, with a larger σ leading to smaller energy gaps. The parameter α is there to avoid the discontinuity at $E_u = E_l$. The corresponding gradient to follow $\mathbf{F}$ is:

$$\mathbf{F} = \frac{\mathbf{g}_l + \mathbf{g}_u}{2} + 2\sigma\left[\frac{E_u - E_l}{E_u - E_l + \alpha} - \frac{1}{2}\left(\frac{E_u - E_l}{E_u - E_l + \alpha}\right)^2\right]\mathbf{d}. \tag{8.103}$$

Note that σ and α might strongly influence the quality (i.e., with the penalty function method the optimization will not converge to the true minimum energy conical intersection point) of the result and the convergence behavior. A large σ and a small α will improve the quality of the result, but make the optimization harder to converge.

8.19 Phase tracking

8.19.1 Phase tracking of the transformation matrix

A Hermitian matrix $\mathbf{H}^{\text{MCH}}$ can always be diagonalized. Its eigenvectors form the rows of a unitary matrix $\mathbf{U}$, which can be used to transform between the original basis and the basis of the eigenfunctions of $\mathbf{H}$.

$$\mathbf{H}^{\text{diag}} = \mathbf{U}^\dagger\mathbf{H}^{\text{MCH}}\mathbf{U}. \tag{8.104}$$

However, the condition that $\mathbf{U}$ diagonalizes $\mathbf{H}^{\text{MCH}}$ is not sufficient to define $\mathbf{U}$ uniquely. Each normalized eigenvector $\mathbf{u}$ can be multiplied by a complex number on the unit circle and still remains a normalized eigenvector.

$$\mathbf{H}\mathbf{u} = h\mathbf{u} \qquad \text{and} \qquad \mathbf{u}^\dagger\mathbf{u} = 1 \tag{8.105}$$

leads to

$$\mathbf{H}\left(\mathrm{e}^{\mathrm{i}\phi}\mathbf{u}\right) = \mathrm{e}^{\mathrm{i}\phi}\left(\mathbf{H}\mathbf{u}\right) = \mathrm{e}^{\mathrm{i}\phi}h\mathbf{u} = h\left(\mathrm{e}^{\mathrm{i}\phi}\mathbf{u}\right) \tag{8.106}$$

and

$$\left(\mathrm{e}^{\mathrm{i}\phi}\mathbf{u}\right)^\dagger\left(\mathrm{e}^{\mathrm{i}\phi}\mathbf{u}\right) = \mathbf{u}^\dagger\mathrm{e}^{-\mathrm{i}\phi}\mathrm{e}^{\mathrm{i}\phi}\mathbf{u} = \mathbf{u}^\dagger\mathbf{u} = 1 \tag{8.107}$$

Thus, for all diagonal matrices $\mathbf{\Phi}$ with elements $\delta_{\beta\alpha}\mathrm{e}^{\mathrm{i}\phi_\beta}$, also the matrix $\mathbf{U}' = \mathbf{U}\mathbf{\Phi}$ diagonalizes $\mathbf{H}^{\text{MCH}}$ (if $\mathbf{U}$ diagonalizes it).

The propagation of the coefficients in the diagonal basis is written as (see section **??**):

$$\mathbf{c}^{\text{diag}}(t + \Delta t) = \underbrace{\mathbf{U}^\dagger(t + \Delta t)\mathbf{R}^{\text{MCH}}(t + \Delta t, t)\mathbf{U}(t)}_{\mathbf{R}^{\text{diag}}(t+\Delta t, t)}\mathbf{c}^{\text{diag}}(t) \tag{8.108}$$

where $\mathbf{U}(t)$ and $\mathbf{U}(t+\Delta t)$ are determined independently from diagonalizing the matrices $\mathbf{H}^{\mathrm{MCH}}(t)$ and $\mathbf{H}^{\mathrm{MCH}}(t+\Delta t)$, respectively. However, depending on the implementation of the diagonalization, $\mathbf{U}(t)$ and $\mathbf{U}(t+\Delta t)$ may carry unrelated, random phases. Even if $\mathbf{H}^{\mathrm{MCH}}(t)$ and $\mathbf{H}^{\mathrm{MCH}}(t+\Delta t)$ were identical, $\mathbf{U}(t)$ and $\mathbf{U}(t+\Delta t)$ might still differ, e.g.:

$$\mathbf{U}(t) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad \text{and} \qquad \mathbf{U}(t+\Delta t) = \begin{pmatrix} \mathrm{i} & 0 \\ 0 & -\mathrm{i} \end{pmatrix} \tag{8.109}$$

The result is that the coefficients **c** pick up random phases during the propagation, leading to random changes in the direction of population transfer, invalidating the whole propagation.

In order to make the phases of $\mathbf{U}(t)$ and $\mathbf{U}(t+\Delta t)$ as similar as possible, SHARC employs a projection technique. First, we define the overlap matrix **V** between $\mathbf{U}(t)$ and $\mathbf{U}(t+\Delta t)$:

$$\mathbf{V} = \mathbf{U}^{\dagger}(t+\Delta t)\mathbf{U}(t) \tag{8.110}$$

For $\Delta t = 0$, clearly

$$\mathbf{U}(t+\Delta t)\mathbf{V} = \mathbf{U}(t) \tag{8.111}$$

and **V** can be identified with the phase matrix $\mathbf{\Phi}$.

For $\Delta t \neq 0$, we must now find a matrix **P** so that

$$\mathbf{U}(t+\Delta t)\mathbf{P} = \mathbf{U}'(t+\Delta t) \tag{8.112}$$

still diagonalizes $\mathbf{H}^{\mathrm{MCH}}(t+\Delta t)$, but which minimizes the phase change with regard to $\mathbf{U}(t)$. The matrix **P** has elements

$$P_{\beta\alpha} = V_{\beta\alpha}\delta\left(E_\beta - E_\alpha\right). \tag{8.113}$$

where E_β is the β-th eigenvalue of $\mathbf{H}^{\mathrm{MCH}}(t+\Delta t)$.

Within the SHARC algorithm, the phase of $\mathbf{U}(t+\Delta t)$ is adjusted to be most similar to $\mathbf{U}(t)$ by calculating first **V**, generating **P** from **V** and the eigenvalues of $\mathbf{H}^{\mathrm{MCH}}(t+\Delta t)$ and calculating the phase-corrected matrix $\mathbf{U}'(t+\Delta t)$ as $\mathbf{U}(t+\Delta t)\mathbf{P}$.

8.19.2 Tracking of the phase of the MCH wave functions

Additionally, within the quantum chemistry programs, the phases of the electronic wave functions may change from one time step to the next one. This will result in changes of the phase of all off-diagonal matrix elements (spin-orbit couplings, transition dipole moments, nonadiabatic couplings). SHARC has several possibilities to correct for that:

- The interface can provide wave function phases through **`QM.out`**.
- If the overlap matrix is available, its diagonal contains the necessary phase information.
- Otherwise the scalar products of old and new nonadiabatic couplings and the relative phase of SOC matrix elements can be used to construct phase information.

8.20 Random initial velocities

Random initial velocities are calculated with a given amount of kinetic energy E per atom a. For each atom, the velocity is calculated as follows, with two uniform random numbers θ and ϕ, from the interval $[0,1[$:

$$\mathbf{v} = \sqrt{2E/m_a}\begin{pmatrix} \cos\theta\sin\phi \\ \sin\theta\sin\phi \\ \cos\phi \end{pmatrix} \tag{8.114}$$

This procedure gives a uniform probability distribution on a sphere with radius $\sqrt{2E/m_a}$.

Note that the translational and rotational components of random initial velocities are not projected out in the current implementation.

Random initial velocities can be requested in the input with **`veloc random`** E, where E is a float defining the kinetic energy per atom (in eV).

8.21 Representations

Within SHARC, two different representations for the electronic states are used. The first is the so-called MCH basis, which is the basis of the eigenfunctions of the molecular Coulomb Hamiltonian. The molecular Coulomb Hamiltonian is the standard electronic Hamiltonian employed by the majority of quantum chemistry programs. It contains only the kinetic energy of the electrons and the potential energy arising from the Coulomb interaction between the electrons and nuclei.

$$\hat{H}_{\text{el}}^{\text{MCH}} = \hat{K}_{\text{e}} + \hat{V}_{\text{ee}} + \hat{V}_{\text{ne}} + \hat{V}_{\text{nn}}. \tag{8.115}$$

With this hamiltonian, states of the same multiplicity couple via the nonadiabatic couplings, while states of different multiplicity do not interact at all.

The second representation used in SHARC is the so-called diagonal representation. It is the basis of the eigenfunctions of the total Hamiltonian.

$$\hat{H}_{\text{el}}^{\text{total}} = \hat{H}_{\text{el}}^{\text{MCH}} + \hat{H}_{\text{el}}^{\text{coup}}. \tag{8.116}$$

The term $\hat{H}_{\text{el}}^{\text{coup}}$ contains additional couplings not contained in the molecular Coulomb Hamiltonian. The most common couplings are spin-orbit couplings and interactions with an external electric field.

$$\hat{H}_{\text{el}}^{\text{coup}} = \hat{H}_{\text{el}}^{\text{SOC}} - \boldsymbol{\mu}\boldsymbol{\epsilon}^{\text{ext}} \tag{8.117}$$

Both of these couplings introduce off-diagonal elements in the total Hamiltonian. Thus, the eigenfunctions of the molecular Coulomb Hamiltonian are not the eigenfunctions of the total Hamiltonian.

Within SHARC, usually quantum chemistry information is read in the MCH representation, while the surface hopping is performed in the diagonal one.

8.21.1 Current state in MCH representation

Oftentimes, it is very useful to know to which MCH state the currently active diagonal state corresponds. If $\hat{H}_{\text{el}}^{\text{coup}}$ is small or the state separation is large, then each diagonal state approximately corresponds to one MCH state. Only in the case of large couplings and/or near-degenerate states are the MCH states strongly mixed in the diagonal states.

In order to obtain for a given time step from the currently active diagonal state β the corresponding MCH state α, a vector $\mathbf{c}^{\text{diag}}$ with $c_i^{\text{diag}} = \delta_{i\beta}$ is generated. The vector is transformed into the MCH representation

$$\mathbf{c}^{\text{MCH}} = \mathbf{U}\mathbf{c}^{\text{diag}}. \tag{8.118}$$

The corresponding MCH state α is the index of the (absolute) largest element of vector $\mathbf{c}^{\text{MCH}}$.

8.22 Sampling from Wigner Distribution

The sampling is based on references [**? ?**].

Besides the equilibrium geometry $\mathbf{R}_{\text{eq}}$, the optimization plus frequency calculation provides a set of vibrational frequencies $\{\nu_i\}$ and the corresponding normal mode vectors $\{\mathbf{n}_i\}$, where i runs from 1 to $N = 3n_{\text{atom}}$.

The normal mode vectors need to be provided in terms of mass-weighted Cartesian normal modes, in atomics units (Bohrs times square root of electron mass, i.e., $a_0 \cdot \sqrt{m_e}$). Most quantum chemistry programs follow different conventions when writing MOLDEN files. MOLPRO and MOLCAS write these files with unweighted Cartesian normal modes, with units of a_0 (Bohrs). GAUSSIAN, TURBOMOLE, Q-CHEM, ADF, and ORCA employ what could be called the "GAUSSIAN convention", which are normalized Cartesian normal modes. COLUMBUS uses yet another convention in the output of their **`suscal.x`** module. The script **`wigner.py`** automatically transforms these different conventions; it does so by applying all possible transformations to the input data until it finds one transformation which produces an orthonormal normal mode matrix. The latter one is then used for the Wigner sampling.

In order to create an initial condition $(\mathbf{R}, \mathbf{v})$, the following procedure is applied. Initially, $\mathbf{R}_0 = \mathbf{R}_{\text{eq}}$ and $\mathbf{v}_0 = 0$. Then, for each normal mode i, two random numbers P_i and Q_i are chosen uniformly from the interval $[-5, 5]$. The value of a ground state quantum Wigner distribution for these values is calculated:

$$W_i = \text{e}^{-(P_i^2 + Q_i^2)}. \tag{8.119}$$

W_i is compared to a uniform random r_i number from $[0,1]$. If $W_i > r_i$, then P_i and Q_i are accepted. Subsequently, the coordinates:

$$\mathbf{R}_i = \mathbf{R}_{i-1} + \frac{Q}{\sqrt{2\nu_i}}\mathbf{n}_i \tag{8.120}$$

and velocities

$$\mathbf{v}_i = \mathbf{v}_{i-1} + \frac{P\sqrt{\nu_i}}{\sqrt{2}}\mathbf{n}_i \tag{8.121}$$

are updated. The random number procedure and updates are repeated for all normal modes, until $(\mathbf{R}_N, \mathbf{v})_N$ is obtained, which constitutes one initial condition. Finally, the center of mass is restored and translational and rotational components are projected out of $\mathbf{v}$. The harmonic potential energy is given by:

$$E_{\text{pot}} = \frac{1}{2}\sum_i \nu_i Q_i^2 \tag{8.122}$$

8.22.1 Sampling at Non-zero Temperature

In the case of a non-zero temperature, the molecule might not be in the vibrational ground state of the harmonic oscillator, but rather in an excited vibrational state. For a given mode i, the probability to be in any given vibrational state j ($j = 0$ is the ground state) is:

$$w_{ij} = \mathrm{e}^{-y\frac{j+1}{2}}\left(\frac{\mathrm{e}^{-\frac{y}{2}}}{1-\mathrm{e}^{-y}}\right)^{-1}, \tag{8.123}$$

where y is ν_i divided by k_BT. In order to find the vibrational state for mode i, a random number is drawn (from $[0,1]$) and used as in equation (**??**).

The displacements and velocity contributions for mode i in state j are then obtained as in equations (**??**) and (**??**), except that the Wigner distribution for state j is calculated as:

$$W_{ij} = (-1)^j \mathrm{e}^{-(P_i^2+Q_i^2)} \sum_m^j (-1)^m \frac{j!}{(j-m)!(m!)^2}\left(2P_i^2 + 2Q_i^2\right)^m . \tag{8.124}$$

8.23 Scaling

The scaling factor (keyword **`scaling`**) applies to all energies and derivatives of energies. Hence, the full Hamiltonian is scaled, and the gradients are scaled. Nothing else is scaled (no dipole moments, nonadiabatic couplings, overlaps, etc).

8.24 Seeding of the RNG

The standard Fortran 90 random number generator (used for **`sharc.x`**, but not for the auxiliary scripts) is seeded by a sequence of integers of length n, where n depends on the computer architecture. The input of SHARC, however, takes only a single RNG seed, which must reproducibly produce the same sequence of random numbers for the same input.

In order to generate the seed sequence from the single input x, the following procedure is applied:

- Query for the number n,
- Generate a first seed sequence $\mathbf{s}$ with $s_i = x + 37i + 17i^2$,
- Seed with the sequence $\mathbf{s}$,
- Obtain a sequence $\mathbf{r}$ of n random numbers on the interval $[0,1[$,
- Generate a second seed sequence $\mathbf{s}'$ with $s'_i = \text{int}\left(65536(r_i - \frac{1}{2})\right)$,
- Reseed with the sequence $\mathbf{s}'$.

The fifth step will generate a sequence of nearly uncorrelated numbers, distributed uniformly over the full range of possible integer values.

8.25 Selection of gradients and nonadiabatic couplings

In order to increase performance, it is possible to omit the calculation of certain gradients and nonadiabatic couplings. An energy-gap-based algorithm selects at each time step a subset of all possible gradients and nonadiabatic couplings to be calculated. Given the diagonal energy E_ξ^{diag} of the current active state ξ, the gradient $\mathbf{g}_\alpha^{\text{MCH}}$ of MCH state α is calculated if:

$$\left|E_\xi^{\text{diag}} - E_\alpha^{\text{MCH}}\right| < \varepsilon_{\text{grad}} \tag{8.125}$$

where $\varepsilon_{\text{grad}}$ is the selection threshold.

Similarly, a nonadiabatic coupling vector $\mathbf{K}_{\beta\alpha}^{\text{MCH}}$ is calculated if:

$$\left|E_\xi^{\text{diag}} - E_\alpha^{\text{MCH}}\right| < \varepsilon_{\text{nac}} \qquad \text{and} \qquad \left|E_\xi^{\text{diag}} - E_\beta^{\text{MCH}}\right| < \varepsilon_{\text{nac}} \tag{8.126}$$

with selection threshold ε_{nac}.

Neither $\mathbf{g}_\alpha^{\text{MCH}}$ nor $\mathbf{K}_{\beta\alpha}^{\text{MCH}}$ are ever calculated if α or β are frozen states.

There is only one keyword (**eselect**) to set the selection threshold, so $\varepsilon_{\text{grad}}$ and ε_{nac} are the same in most cases.

8.26 State ordering

The canonical ordering of MCH states of different S and M_S in SHARC is as follows. In the innermost loop, the quantum number is increased; then M_S and finally S. Example:

```
nstates 3 0 3
```

In this example, the order of states is given as:

Number	Label	S	M_S	n
1	S_0	0	0	1
2	S_1	0	0	2
3	S_2	0	0	3
4	T_1^-	2	-1	1
5	T_2^-	2	-1	2
6	T_3^-	2	-1	3
7	T_1^0	2	0	1
8	T_2^0	2	0	2
9	T_3^0	2	0	3
10	T_1^+	2	+1	1
11	T_2^+	2	+1	2
12	T_3^+	2	+1	3

The canonical ordering of states is for example important in order to specify the initial state in the MCH basis (using the **state** keyword in the input file).

Note that the diagonal states do not follow the same prescription. Since the diagonal states are in general not eigenfunctions of the total spin operator, they do not have a well-defined multiplicity. Hence, the diagonal states are simply ordered by increasing energy.

8.27 Surface Hopping

Given two coefficient vectors $\mathbf{c}^{\text{diag}}(t)$ and $\mathbf{c}^{\text{diag}}(t+\Delta t)$ and the corresponding propagator matrix $\mathbf{R}^{\text{diag}}(t+\Delta t, t)$, the surface hopping probabilities are given by

$$P_{\beta\to\alpha} = \left(1 - \frac{\left|c_\beta^{\text{diag}}(t+\Delta t)\right|^2}{\left|c_\beta^{\text{diag}}(t)\right|^2}\right) \times \frac{\Re\left[c_\alpha^{\text{diag}}(t+\Delta t)R^*_{\alpha\beta}\left(c_\beta^{\text{diag}}(t)\right)^*\right]}{\left|c_\beta^{\text{diag}}(t)\right|^2 - \Re\left[c_\beta^{\text{diag}}(t+\Delta t)R^*_{\beta\beta}\left(c_\beta^{\text{diag}}(t)\right)^*\right]}. \tag{8.127}$$

where, however, $P_{\beta\to\beta} = 0$ and all negative $P_{\beta\to\alpha}$ are set to zero. This equation is the default in SHARC, and can be used with **`hopping_procedure sharc`**.

Alternatively, the hopping probabilities can be obtained with the "global flux surface hopping" method by Prezhdo and coworkers [**?**]. The equation is:

$$P_{\beta\to\alpha} = \left(1 - \frac{\left|c_\beta^{\text{diag}}(t+\Delta t)\right|^2}{\left|c_\beta^{\text{diag}}(t)\right|^2}\right) \times \frac{|c_\alpha^{\text{diag}}(t+\Delta t)|^2 - |c_\alpha^{\text{diag}}(t)|^2}{\sum_i \max\left[0, -(|c_i^{\text{diag}}(t+\Delta t)|^2 - |c_i^{\text{diag}}(t)|^2)\right]}. \tag{8.128}$$

As above, $P_{\beta\to\beta} = 0$ and all negative $P_{\beta\to\alpha}$ are set to zero. This equation and can be used with **`hopping_procedure gfsh`**.

In any case, the hopping procedure itself obtains a uniform random number r from the interval $[0,1]$. A hop to state α is performed, if

$$\sum_{i=1}^{\alpha-1} P_{\beta\to i} < r \leq P_{\beta\to\alpha} + \sum_{i=1}^{\alpha-1} P_{\beta\to i} \tag{8.129}$$

See section **??** for further details on how frustrated hops (hops according to the hopping probabilities, but where not enough energy is available to execute the hop) are handled.

8.28 Velocity Verlet

The nuclear coordinates of atom A are updated according to the Velocity Verlet algorithm [**?**], based on the gradient of state β at $\mathbf{R}(t)$ and $\mathbf{R}(t+\Delta t)$:

$$\mathbf{a}_A(t) = -\frac{1}{m_A}\nabla_{\mathbf{R_A}} E_\beta(\mathbf{R}(t)) \tag{8.130}$$

$$\mathbf{a}_A(t+\Delta t) = -\frac{1}{m_A}\nabla_{\mathbf{R_A}} E_\beta(\mathbf{R}(t+\Delta t)) \tag{8.131}$$

$$\mathbf{R}_A(t+\Delta t) = \mathbf{R}_A(t) + \mathbf{v}_A(t)\Delta t + \frac{1}{2}\mathbf{a}_A(t)\Delta t^2 \tag{8.132}$$

$$\mathbf{v}_A(t+\Delta t) = \mathbf{v}_A(t) + \frac{1}{2}\left[\mathbf{a}_A(t) + \mathbf{a}_A(t+\Delta t)\right]\Delta t \tag{8.133}$$

Currently, there are no other integrators for the nuclear motion implemented in SHARC.

8.29 Wavefunction propagation

The electronic wave function is needed in order to carry out surface hopping. The electronic wave function is expanded in the basis of the so-called model space $\mathcal{S}$, which includes the few lowest states $|\psi_\alpha^{\text{MCH}}\rangle$ of the multiplicities under consideration (e.g. the 3 lowest singlet and 2 lowest triplet states).

$$\Psi_{\text{el}}(t) = \sum_{\alpha\in\mathcal{S}} c_\alpha^{\text{MCH}} \left|\psi_\alpha^{\text{MCH}}\right\rangle \tag{8.134}$$

All multiplet components are included explicitly, i.e., the inclusion of an MCH triplet state adds three explicit states to the model space (the three components of the triplet).

Within SHARC, the wave function is represented just by the vector $\mathbf{c}^{\mathrm{MCH}}$. The Hamiltonian $\mathbf{H}^{\mathrm{MCH}}$ is represented in matrix form with elements:

$$H^{\mathrm{MCH}}_{\beta\alpha} = \left\langle \psi^{\mathrm{MCH}}_{\beta} \middle| \hat{H}^{\mathrm{total}}_{\mathrm{el}} \middle| \psi^{\mathrm{MCH}}_{\alpha} \right\rangle \tag{8.135}$$

From the MCH representation, the diagonal representation can be obtained by unitary transformation within the model space $\mathcal{S}$ ($\mathbf{U}^\dagger\mathbf{H}^{\mathrm{MCH}}\mathbf{U} = \mathbf{H}^{\mathrm{diag}}$ and $\mathbf{U}^\dagger\mathbf{c}^{\mathrm{MCH}} = \mathbf{c}^{\mathrm{diag}}$):

$$\Psi_{\mathrm{el}}(t) = \sum_{\alpha\in\mathcal{S}} c^{\mathrm{diag}}_{\alpha} \left|\psi^{\mathrm{diag}}_{\alpha}\right\rangle \tag{8.136}$$

and

$$H^{\mathrm{diag}}_{\beta\alpha} = \left\langle \psi^{\mathrm{diag}}_{\beta} \middle| \hat{H}^{\mathrm{total}}_{\mathrm{el}} \middle| \psi^{\mathrm{diag}}_{\alpha} \right\rangle \tag{8.137}$$

The propagation of the electronic wave function from time t to $t+\Delta t$ can then be written as the product of a propagation matrix with the coefficients at time t:

$$\mathbf{c}^{\mathrm{diag}}(t+\Delta t) = \mathbf{R}^{\mathrm{diag}}(t+\Delta t, t)\mathbf{c}^{\mathrm{diag}}(t) \tag{8.138}$$

or

$$\mathbf{c}^{\mathrm{diag}}(t+\Delta t) = \underbrace{\mathbf{U}^\dagger(t+\Delta t)\mathbf{R}^{\mathrm{MCH}}(t+\Delta t, t)\mathbf{U}(t)}_{\mathbf{R}^{\mathrm{diag}}(t+\Delta t,t)}\mathbf{c}^{\mathrm{diag}}(t) \tag{8.139}$$

In order to calculate $\mathbf{R}^{\mathrm{MCH}}(t+\Delta t, t)$, SHARC uses (unitary) operator exponentials.

8.29.1 Propagation using nonadiabatic couplings

Here we assume that in the dynamics the interaction between the electronic states is described by a matrix of nonadiabatic couplings $\mathbf{K}^{\mathrm{MCH}}(t)$, such that

$$\left(\mathbf{K}^{\mathrm{MCH}}(t)\right)_{\beta\alpha} = \left\langle \psi_\beta(t) \middle| \frac{\partial}{\partial t} \middle| \psi_\alpha(t) \right\rangle \tag{8.140}$$

or

$$\left(\mathbf{K}^{\mathrm{MCH}}(t)\right)_{\beta\alpha} = \frac{\partial\mathbf{R}}{\partial t} \cdot \left\langle \psi_\beta(t) \middle| \frac{\partial}{\partial\mathbf{R}} \middle| \psi_\alpha(t) \right\rangle. \tag{8.141}$$

In equation (**??**), the time-derivative couplings are directly calculated by the quantum chemistry program (use `coupling ddt` in the SHARC input), while in (**??**) the matrix $\mathbf{K}^{\mathrm{MCH}}(t)$ is obtained from the scalar product of the nuclear velocity and the nonadiabatic coupling vectors (use `coupling ddr` in the input).

The propagation matrix can then be written as

$$\mathbf{R}^{\mathrm{MCH}}(t+\Delta t, t) = \hat{\mathfrak{T}} \exp\left[-\int_t^{t+\Delta t} \left(\frac{\mathrm{i}}{\hbar}\mathbf{H}^{\mathrm{MCH}}(\tau) + \mathbf{K}^{\mathrm{MCH}}(\tau)\right) \mathrm{d}\tau\right] \tag{8.142}$$

with the time-ordering operator $\hat{\mathfrak{T}}$. For small time steps Δt, $\mathbf{H}^{\mathrm{MCH}}(\tau)$ and $\mathbf{K}^{\mathrm{MCH}}(\tau)$ can be interpolated linearly

$$\mathbf{R}^{\mathrm{MCH}}(t+\Delta t, t) = \exp\left[-\frac{1}{2}\left(\frac{\mathrm{i}}{\hbar}\mathbf{H}^{\mathrm{MCH}}(t) + \frac{\mathrm{i}}{\hbar}\mathbf{H}^{\mathrm{MCH}}(t+\Delta t) + \mathbf{K}^{\mathrm{MCH}}(t) + \mathbf{K}^{\mathrm{MCH}}(t+\Delta t)\right)\Delta t\right] \tag{8.143}$$

And in order to have a sufficiently small time step for this to work, the interval $(t, t+\Delta t)$ is further split into subtime steps $\Delta\tau = \frac{\Delta t}{n}$.

$$\mathbf{R}^{\mathrm{MCH}}(t+\Delta t, t) = \prod_{i=1}^{n} \mathbf{R}_i \tag{8.144}$$

$$\mathbf{R}_i = \exp\left[-\left(\frac{\mathrm{i}}{\hbar}\mathbf{H}_i + \mathbf{K}_i\right)\Delta\tau\right] \tag{8.145}$$

$$\mathbf{H}^{\mathrm{MCH}}(t) + \frac{i}{n}\left(\mathbf{H}^{\mathrm{MCH}}(t+\Delta t) - \mathbf{H}^{\mathrm{MCH}}(t)\right) \tag{8.146}$$

$$\mathbf{K}_i = \mathbf{K}^{\mathrm{MCH}}(t) + \frac{i}{n}\left(\mathbf{K}^{\mathrm{MCH}}(t+\Delta t) - \mathbf{K}^{\mathrm{MCH}}(t)\right) \tag{8.147}$$

8.29.2 Propagation using overlap matrices

In many situations, the nonadiabatic couplings in $\mathbf{K}^{\mathrm{MCH}}$ are very localized on the potential hypersurfaces. If this is the case, in the dynamics very short time steps are necessary to properly sample the nonadiabatic couplings. If too large time steps are used, part of the coupling may be missed, leading to wrong population transfer. The local diabatization algorithm gives more numerical stability in these situations. It can be requested with the line **`coupling overlap`** in the input file.

Within this algorithm, the change of the electronic states between time steps is described by the overlap matrix $\mathbf{S}^{\mathrm{MCH}}(t, t+\Delta t)$

$$\left(\mathbf{S}^{\mathrm{MCH}}(t, t+\Delta t)\right)_{\beta\alpha} = \left\langle \psi_\beta(t) \middle| \psi_\alpha(t+\Delta t) \right\rangle \tag{8.148}$$

With this, the propagator matrix can be written as

$$\mathbf{R}^{\mathrm{MCH}}(t+\Delta t, t) = \mathbf{S}^{\mathrm{MCH}}(t, t+\Delta t)^\dagger \prod_{i=1}^{n} \mathbf{R}_i \tag{8.149}$$

$$\mathbf{R}_i = \exp\left[-\frac{\mathrm{i}}{\hbar}\mathbf{H}_i \Delta\tau\right] \tag{8.150}$$

$$\mathbf{H}_i = \mathbf{H}^{\mathrm{MCH}}(t) + \frac{i}{n}\left(\mathbf{H}^{\mathrm{MCH}}_{\mathrm{tra}} - \mathbf{H}^{\mathrm{MCH}}(t)\right) \tag{8.151}$$

$$\mathbf{H}^{\mathrm{MCH}}_{\mathrm{tra}} = \mathbf{S}^{\mathrm{MCH}}(t, t+\Delta t)\mathbf{H}^{\mathrm{MCH}}(t+\Delta t)\mathbf{S}^{\mathrm{MCH}}(t, t+\Delta t)^\dagger \tag{8.152}$$

List of Tables

List of Figures