MRChem Documentation

Stig Rune Jensen, Luca Frediani, Peter Wind

Jan 17, 2024

CONTENTS

MRChem is a numerical real-space code for molecular electronic structure calculations within the self-consistent field (SCF) approximations of quantum chemistry (Hartree-Fock and Density Functional Theory). The code is divided in two main parts: the MultiResolution Computation Program Package (MRCPP), which is a general purpose numerical mathematics library based on multiresolution analysis and the multiwavelet basis which provide low-scaling algorithms as well as rigorous error control in numerical computations, and the MultiResolution Chemistry (MRChem) program that uses the functionalities of MRCPP for computational chemistry applications.

The code is being developed at the Hylleraas Centre for Quantum Molecular Sciences at UiT - The Arctic University of Norway.

The code is under active development, and the latest stable releases as well as development versions can be found on GitHub.

CHAPTER

ONE

FEATURES IN MRCHEM-1.1:

- **Wave functions:**
  - **Kohn-Sham DFT**
    - Spin-polarized
    - Spin-unpolarized
    - LDA, GGA and hybrid functionals
  - **Hartree-Fock**
    - Restricted closed-shell
    - Unrestricted
  - **Explicit external fields**
    - Electric field
  - **Solvent effects**
    - Cavity-free PCM
- **Properties:**
  - Ground state energy
  - Dipole moment
  - Quadrupole moment
  - Polarizability
  - Magnetizability
  - NMR shielding constant
  - Geometric derivative
- **Parallel implementation:**
  - Shared memory (OpenMP): ~20 cores
  - Distributed memory (MPI): ~1000 procs
  - Hybrid scheme (MPI + OpenMP): ~10 000 cores
- **Current size limitations:**
  - ~2000 orbitals on ~100 high-end compute nodes (128 core/256GiB mem)
  - ~100 orbitals on a single high-memory (1TB) compute node

CHAPTER

TWO

UPCOMING FEATURES:

- **Wave functions:**
  - Meta-GGAs
  - ZORA Hamiltonian
  - Periodic Boundary Conditions
  - External magnetic field
- **Properties:**
  - Optical rotation
  - Spin-spin coupling constant
  - Hyperfine coupling constant
  - Magnetically induced currents
  - Hyperpolarizability
  - Geometry optimization
- **Performance:**
  - Reduced memory footprint
  - Improved DFT scaling and performance

2.1 Installation

2.1.1 Build prerequisites

- Python-3.7 (or later)
- CMake-3.14 (or later)
- GNU-5.4 or Intel-17 (or later) compilers (C++14 standard)

Hint: We have collected the recommended modules for the different Norwegian HPC systems under `tools/<machine>.env`. These files can be sourced in order to get a working environment on the respective machines, and may also serve as a guide for other HPC systems.

C++ dependencies

The MRChem program depends on the following C++ libraries:

- Input handling: nlohmann/json-3.6
- Multiwavelets: MRCPP-1.4
- Linear algebra: Eigen-3.4
- DFT functionals: XCFun-2.0

All these dependencies will be downloaded automatically at configure time by CMake, but can also be linked manually by setting the variables:

```
MRCPP_DIR=<path_to_mrcpp>/share/cmake/MRCPP
XCFun_DIR=<path_to_xcfun>/share/cmake/XCFun
Eigen3_DIR=<path_to_eigen3>/share/eigen3/cmake
nlohmann_json_DIR=<path_to_nlohmann_json>
```

Python dependencies

Users only need a Python3 interpreter, which is used for configuration (`setup` script) as well as launching the program (`mrchem` script).

Developers will need some extra Python packages to update the input parser and build the documentation locally with Sphinx.

We **strongly** suggest not to install these Python dependencies globally, but rather to use a local virtual environment. We provide a `Pipfile` for specifying the Python dependencies. We recommend using Pipenv, since it manages virtual environment and package installation seamlessly. After installing it with your package manager, run:

```
$ pipenv install --dev
```

to create a virtual environment with all developer packages installed.

The environment can be activated with:

```
$ pipenv shell
```

Alternatively, any Python command can be run within the virtual environment by doing:

```
$ pipenv run python -c "print('Hello, world')"
```

2.1.2 Obtaining and building the code

The latest development version of MRChem can be found on the `master` branch on GitHub:

```
$ git clone https://github.com/MRChemSoft/mrchem.git
```

The released versions can be found from Git tags `vX.Y.Z` under the `release/X.Y` branches in the same repository, or a zip file can be downloaded from Zenodo.

By default, all dependencies will be **fetched** at configure time if they are not already available.

Configure

The `setup` script will create a directory called `<build-dir>` and run CMake. There are several options available for the setup, the most important being:

`--cxx=<CXX>`
C++ compiler [default: g++]

`--omp`
Enable OpenMP parallelization [default: False]

`--mpi`
Enable MPI parallelization [default: False]

`--type=<TYPE>`
Set the CMake build type (debug, release, relwithdebinfo, minsizerel) [default: release]

`--prefix=<PATH>`
Set the install path for make install [default: '/usr/local']

`--cmake-options=<STRING>`
Define options to CMake [default: '']

`-h --help`
List all options

The code can be built with four levels of parallelization:

- no parallelization
- only shared memory (OpenMP)
- only distributed memory (MPI)
- hybrid OpenMP + MPI

Note: In practice we recommend the **shared memory version** for running on your personal laptop/workstation, and the **hybrid version** for running on a HPC cluster. The serial and pure MPI versions are only useful for debugging.

The default build is *without* parallelization and using GNU compilers:

```
$ ./setup --prefix=<install-dir> <build-dir>
```

To use Intel compilers you need to specify the `--cxx` option:

```
$ ./setup --prefix=<install-dir> --cxx=icpc <build-dir>
```

To build the code with shared memory (OpenMP) parallelization, add the `--omp` option:

```
$ ./setup --prefix=<install-dir> --omp <build-dir>
```

To build the code with distributed memory (MPI) parallelization, add the `--mpi` option *and* change to the respective MPI compilers (`--cxx=mpicxx` for GNU and `--cxx=mpiicpc` for Intel):

```
$ ./setup --prefix=<install-dir> --omp --mpi --cxx=mpicxx <build-dir>
```

When dependencies are fetched at configuration time, they will be downloaded into `<build-dir>/_deps`. For the example of MRCPP, sources are saved into the folders `<build-dir>/_deps/mrcpp_sources-src` and built into `<build-dir>/_deps/mrcpp_sources-build`.

Note: If you compile the MRCPP library manually as a separate project, the level of parallelization **must be the same** for MRCPP and MRChem. Similar options apply for the MRCPP setup, see mrcpp.readthedocs.io.

Build

If the CMake configuration is successful, the code is compiled with:

```
$ cd <build-dir>
$ make
```

Test

A test suite is provided to make sure that everything compiled properly. To run a collection of small unit tests:

```
$ cd <build-dir>
$ ctest -L unit
```

To run a couple of more involved integration tests:

```
$ cd <build-dir>
$ ctest -L integration
```

Install

After the build has been verified with the test suite, it can be installed with the following command:

```
$ cd <build-dir>
$ make install
```

This will install *two* executables under the `<install-path>`:

```
<install-path>/bin/mrchem       # Python input parser and launcher
<install-path>/bin/mrchem.x     # MRChem executable
```

Please refer to the *User's Manual* for instructions for how to run the program.

Hint: We have collected scripts for configure and build of the hybrid OpenMP + MPI version on the different Norwegian HPC systems under `tools/<machine>.sh`. These scripts will build the current version under `build-${version}`, run the unit tests and install under `install-${version}`, e.g. to build version v1.0.0 on Fram:

```
$ cd mrchem
$ git checkout v1.0.0
$ tools/fram.sh
```

The configure step requires internet access, so the scripts must be run on the login nodes, and it will run on a single core, so it might take some minutes to complete. The scripts will *not* install the *Python dependencies*, so this must be done manually in order to run the code.

2.2 User's Manual

The MRChem program comes as two executables:

```
<install-path>/bin/mrchem                    # Python input parser and launcher
<install-path>/bin/mrchem.x                  # MRChem main executable
```

where the former is a Python script that reads and validates the *user input file* and produces a new *program input file* which is then passed as argument to the latter, which is the actual C++ executable.

The input and output of the program is thus organized as *three* separate files:

File extension	Description	Format
`.inp`	User input file	GETKW/JSON
`.json`	Program input/output	JSON
`.out`	User output file	Text

The name of the user input file can be anything, as long as it has the `.inp` extension, and the corresponding `.json` and `.out` files will get the same name prefix. The JSON program file will get both an `"input"` and an `"output"` section. This `"input"` section is rather detailed and contains very implementation specific keywords, but it is automatically generated by the `mrchem` script, based on the more generic keywords of the user input file. The `mrchem` script will further launch the `mrchem.x` main executable, which will produce the text output file as well as the `"output"` section of the JSON in/out file. The contents of all these files will be discussed in more detail in the sections below.

2.2.1 Running the program

In the following we will assume to have a valid user input file for the water molecule called `h2o.inp`, e.g. like this

```
world_prec = 1.0e-4

WaveFunction {
  method = B3LYP
}

Molecule {
$coords
O  0.0000  0.000 -0.125
H -1.4375  0.000  1.025
H  1.4375  0.000  1.025
$end
}
```

To run the calculation, pass the file name (without extension) as argument to the `mrchem` script (make sure you understand the difference between the `.inp`, `.json` and `.out` file, as described in the previous section):

```
$ mrchem h2o
```

This will under the hood actually do the following two steps:

```
$ mrchem h2o.inp > h2o.json
$ mrchem.x h2o.json > h2o.out
```

The first step includes input validation, which means that everything that passes this step is a well-formed computation.

Dry-running the input parser

The execution of the two steps above can be done separately by dry-running the parser script:

```
$ mrchem --dryrun h2o
```

This will run only the input validation part and generate the `h2o.json` program input, but it will *not* launch the main executable `mrchem.x`. This can then be done manually in a subsequent step by calling:

```
$ mrchem.x h2o.json
```

This separation can be useful for instance for developers or advanced users who want to change some automatically generated input values before launching the actual program, see *Input schema*.

Printing to standard output

By default the program will write to the text output file (`.out` extension), but if you rather would like it printed in the terminal you can add the `--stdout` option (then no text output file is created):

```
$ mrchem --stdout h2o
```

Reproducing old calculations

The JSON in/out file acts as a full record of the calculation, and can be used to reproduce old results. Simply pass the JSON file once more to `mrchem.x`, and the `"output"` section will be overwritten:

```
$ mrchem.x h2o.json
```

User input in JSON format

The user input file can be written in JSON format instead of the standard syntax which is described in detail below. This is very convenient if you have for instance a Python script to generate input files. The water example above in JSON format reads (the `coords` string is not very elegant, but unfortunately that's just how JSON works...):

```
{
  "world_prec": 1.0e-4,
  "WaveFunction": {
    "method": "B3LYP"
  },
  "Molecule": {
    "coords": "O  0.0000  0.000 -0.125\nH -1.4375  0.000  1.025\nH  1.4375  0.000  1.025\
↪n"
  }
}
```

which can be passed to the input parser with the `--json` option:

```
$ mrchem --json h2o
```

Note: A *user input file* in JSON format must **NOT** be confused with the JSON in/out file for the `mrchem.x` program. The file should still have a `.inp` extension, and contain all the same keywords which have to be validated and translated by the `mrchem` script into the `.json` *program input file*.

Parallel execution

The MRChem program comes with support for both shared memory and distributed memory parallelization, as well as a hybrid combination of the two. In order to activate these capabilities, the code needs to be compiled with OpenMP and/or MPI support (`--omp` and/or `--mpi` options to the CMake `setup` script, see *Installation* instructions).

Shared memory OpenMP

For the shared memory part, the program will automatically pick up the number of threads from the environment variable `OMP_NUM_THREADS`. If this variable is *not* set it will usually default to the maximum available. So, to run the code on 16 threads (all sharing the same physical memory space):

```
$ OMP_NUM_THREADS=16 mrchem h2o
```

Distributed memory MPI

In order to run a program in an MPI parallel fashion, it must be executed with an MPI launcher like `mpirun`, `mpiexec`, `srun`, etc. Note that it is only the main executable `mrchem.x` that should be launched in parallel, **not** the `mrchem` input parser script. This can be achieved *either* by running these separately in a dry-run (here two MPI processes):

```
$ mrchem --dryrun h2o
$ mpirun -np 2 mrchem.x h2o.json
```

or in a single command by passing the launcher string as argument to the parser:

```
$ mrchem --launcher="mpirun -np 2" h2o
```

This string can contain any argument you would normally pass to `mpirun` as it will be literally prepended to the `mrchem.x` command when the `mrchem` script executes the main program.

Hint: For best performance, it is recommended to use shared memory *within* each NUMA domain (usually one per socket) of your CPU, and MPI across NUMA domains and ultimately machines. Ideally, the number of OpenMP threads should be between 8-20. E.g. on hardware with two sockets of 16 cores each, use OMP_NUM_THREADS=16 and scale the number of MPI processes by the size of the molecule, typically one process per ~5 orbitals or so (and definitely not *more* than one process per orbital).

Job example (Betzy)

This job will use 4 compute nodes, with 12 MPI processes on each, and the MPI process will use up to 15 OpenMP threads. 4 MPI process per node are used for the "Bank". The Bank processes are using only one thread, therefore there is in practice no overallocation. It is however important that bank_size is set to be at least 4*4 = 16 (it is by default set, correctly, to one third of total MPI size, i.e. 4*12/3=16). It would also be possible to set 16 tasks per node, and set the bank size parameter accordingly to 8*4=32. The flags are optimized for the OpenMPI (foss) library on Betzy (note that H2O is a very small molecule for such setup!).

```
#!/bin/bash -l
#SBATCH --nodes=4
#SBATCH --tasks-per-node=12

export UCX_LOG_LEVEL=ERROR
export OMP_NUM_THREADS=15

~/my_path/to/mrchem --launcher='mpirun --rank-by node --map-by socket --bind-to numa --
↪oversubscribe' h2o
```

--rank-by node
: Tells the system to place the first MPI rank on the first node, the second MPI rank on the second node, until the last node, then start at the first node again.

--map-by socket
: Tells the system to map (group) MPI ranks according to socket before distribution between nodes. This will ensure that for example two bank cores will access different parts of memory and be placed as the 16th thread of a numa group.

--bind-to numa
: Tells the system to bind cores to one NUMA (Non Uniform Memory Access) group. On Betzy memory configuration groups cores by groups of 16, with cores in the same group having the same access to memory (other cores will have access to that part of the memory too, but slower). That means that a process will only be allowed to use one of the 16 cores of the group. (The operating system may change the core assigned to a thread/process and, without precautions, it may be assigned to any other core, which would result in much reduced performance). The 16 cores of the group may then be used by the threads initiated by that MPI process.

--oversubscribe
: To tell MPI that it is should accept that the number of MPI processes times the number of threads is larger than the number of available cores.

Advanced option: Alternatively one can get full control of task placement using the Slurm workload manager by replacing `mpirun` with `srun` and setting explicit CPU masks as:

```
~/my_path/to/mrchem --launcher='srun --cpu-bind=mask_cpu:0xFFFE00000000, \\
0xFFFE0000000000000000000000000000,0xFFFE000000000000,0xFFFE00000000000000000000000000000000, \\
0xFFFE,0xFFFE00000000000000000,0xFFFE0000,0xFFFE0000000000000000000000,0x10000, \\
0x10000000000000000000,0x100000000,0x10000000000000000000000000 \\
--distribution=cyclic:cyclic' h2o
```

`--cpu-bind=mask_cpu:0xFFFE00000000,0xFFFE0000000000000000000000000000,...` give the core (or cpu) masks the process have access to. 0x means that the number is in hexadecimal. For example `0xFFFE00000000` is `111111111111110000000000000000000000000000000000` in binary, meaning that the first process can not use the first 33 cores, then it can use the cores from position 34 up to position 48, and nothing else.

`--distribution=cyclic:cyclic` The first cyclic will put the first rank on the first node, the second rank on the second node etc. The second cyclic distribute the ranks withing the nodes.

More examples can be found in the mrchem-examples repository on GitHub.

Parallel pitfalls

Warning: Parallel program execution is not a black box procedure, and the behavior and efficiency of the run depends on several factors, like hardware configuration, operating system, compiler type and flags, libraries for OpenMP and MPI, type of queing system on a shared cluster, etc. Please make sure that the program runs correctly on *your* system and is able to utilize the computational resources before commencing production calculations.

Typical pitfalls for OpenMP

- Not compiling with correct OpenMP support.
- Not setting number of threads correctly.
- **Hyper-threads:** the round-robin thread distribution might fill all hyper-threads on each core before moving on to the next physical core. In general we discourage the use of hyper-threads, and recommend a single thread per physical core.
- **Thread binding:** all threads may be bound to the same core, which means you can have e.g. 16 threads competing for the limited resources available on this single core (typically two hyper-threads) while all other cores are left idle.

Typical pitfalls for MPI

- Not compiling with the correct MPI support.
- Default launcher options might not give correct behavior.
- **Process binding:** if a process is bound to a core, then all its spawned threads will also be bound to the same core. In general we recommend binding to socket/NUMA.
- **Process distribution:** in a multinode setup, all MPI processes might land on the same machine, or the round-robin procedure might count each core as a separate machine.

How to verify a parallel MRChem run

- In the printed output, verify that MRCPP has actually been compiled with correct support for MPI and/or OpenMP:

```
------------------------------------------------------------------------

MRCPP version         : 1.2.0
Git branch            : master
Git commit hash       : 686037cb78be601ac58b
Git commit author     : Stig Rune Jensen
Git commit date       : Wed Apr 8 11:35:00 2020 +0200

Linear algebra        : EIGEN v3.3.7
Parallelization       : MPI/OpenMP

------------------------------------------------------------------------
```

- In the printed output, verify that the correct number of processes and threads has been detected:

```
-------------------------------------------------------------------------

 MPI processes         :      (no bank)                                 2
 OpenMP threads        :                                               16
 Total cores           :                                               32

-------------------------------------------------------------------------
```

- Monitor your run with `top` to see that you got the expected number of `mrchem.x` processes (MPI), and that they actually run at the expected CPU percentage (OpenMP):

```
PID   USER      PR  NI    VIRT    RES    SHR S   %CPU  %MEM     TIME+ COMMAND
9502  stig      25   5  489456 162064   6628 R 1595,3   2,0   0:14.50 mrchem.x
9503  stig      25   5  489596 162456   6796 R 1591,7   2,0   0:14.33 mrchem.x
```

- Monitor your run with `htop` to see which core/hyper-thread is being used by each process. This is very useful to get the correct binding/pinning of processes and threads. In general you want one threads per core, which means that every other hyper-thread should remain idle. In a hybrid MPI/OpenMP setup it is rather common that each MPI process becomes bound to a single core, which means that all threads spawned by this process will occupy the same core (possibly two hyper-threads). This is then easily detected with `htop`.
- Perform dummy executions of your parallel launcher (`mpirun`, `srun`, etc) to check whether it picks up the correct parameters from the resource manager on your cluster (SLURM, Torque, etc). You can then for instance report bindings and host name for each process:

```
$ mpirun --print-rank-map hostname
```

  Play with the launcher options until you get it right. Note that Intel and OpenMPI have slightly different options for their `mpirun` and usually different behavior. Beware that the behavior can also change when you move from single- to multinode execution, so it is in general not sufficient to verify you runs on a single machine.
- Perform a small scaling test on e.g. 1, 2, 4 processes and/or 1, 2, 4 threads and verify that the total computation time is reduced as expected (don't expect 100% efficiency at any step).

2.2.2 User input file

The input file is organized in sections and keywords that can be of different type. Input keywords and sections are **case-sensitive**, while *values* are **case-insensitive**.

```
Section {
  keyword_1 = 1                         # int
  keyword_2 = 3.14                      # float
  keyword_3 = [1, 2, 3]                 # int array
  keyword_4 = foo                       # string
  keyword_5 = true                      # boolean
}
```

Valid options for booleans are `true/false`, `on/off` or `yes/no`. Single word strings can be given without quotes (be careful of special characters, like slashes in file paths). A complete list of available input keywords can be found in the *User input reference*.

Top section

The main input section contain four keywords: the relative precision ϵ_{rel} that will be guaranteed in the calculation and the size, origin and unit of the computational domain. The top section is not specified by name, just write the keywords directly, e.g

```
world_prec = 1.0e-5                    # Overall relative precision
world_size = 5                         # Size of domain 2^{world_size}
world_unit = bohr                      # Global length unit
world_origin = [0.0, 0.0, 0.0]         # Global gauge origin
```

The relative precision sets an upper limit for the number of correct digits you are expected to get out of the computation (note that $\epsilon_{rel} = 10^{-6}$ yields μ Ha accuracy for the hydrogen molecule, but only mHa accuracy for benzene).

The computational domain is always symmetric around the origin, with *total* size given by the `world_size` parameter as $[2^n]^3$, e.i. `world_size = 5` gives a domain of $[-16, 16]^3$. Make sure that the world is large enough to allow the molecular density to reach zero on the boundary. The `world_size` parameter can be left out, in which case the size will be estimated based on the molecular geometry. The `world_unit` relates to **all** coordinates given in the input file and can be one of two options: `angstrom` or `bohr`.

Note: The `world_size` will be only approximately scaled by the angstrom unit, by adding an extra factor of 2 rather than the appropriate factor of ~1.89. This means that e.g. `world_size = 5` ($[-16, 16]^3$) with `world_unit = angstrom` will be translated into $[-32, 32]^3$ bohrs.

Precisions

MRChem uses a smoothed nuclear potential to avoid numerical problems in connection with the $Z/|r - R|$ singularity. The smoothing is controlled by a single parameter `nuc_prec` that is related to the expected error in the energy due to the smoothing. There are also different precision parameters for the *construction* of the Poisson and Helmholtz integral operators.

```
Precisions {
  nuclear_prec = 1.0e-6                # For construction of nuclear potential
  poisson_prec = 1.0e-6                # For construction of Poisson operators
  helmholtz_prec = 1.0e-6              # For construction of Helmholtz operatos
}
```

By default, all precision parameters follow `world_prec` and usually don't need to be changed.

Printer

This section controls the format of the printed output file (`.out` extension). The most important option is the `print_level`, but it also gives options for number of digits in the printed output, as well as the line width (defaults are shown):

```
Printer {
  print_level = 0                      # Level of detail in the printed output
  print_width = 75                     # Line width (in characters) of printed output
  print_prec = 6                       # Number of digits in floating point output
}
```

Note that energies will be printed with *twice* as many digits. Available print levels are:

- `print_level=-1` no output is printed
- `print_level=0` prints mainly properties
- `print_level=1` adds timings for individual steps
- `print_level=2` adds memory and timing information on `OrbitalVector` level
- `print_level=3` adds details for individual terms of the Fock operator
- `print_level=4` adds memory and timing information on `Orbital` level
- `print_level>=5` adds debug information at MRChem level
- `print_level>=10` adds debug information at MRCPP level

MPI

This section defines some parameters that are used in MPI runs (defaults shown):

```
MPI {
  bank_size = -1                        # Number of processes used as memory bank
  numerically_exact = false             # Guarantee MPI invariant results
  share_nuclear_potential = false       # Use MPI shared memory window
  share_coulomb_potential = false       # Use MPI shared memory window
  share_xc_potential = false            # Use MPI shared memory window
}
```

The memory bank will allow larger molecules to get though if memory is the limiting factor, but it will be slower, as the bank processes will not take part in any computation. For calculations involving exact exchange (Hartree-Fock or hybrid DFT functionals) a memory bank is **required** whenever there's more than one MPI process. A negative bank size will set it automatically based on the number of available processes. For pure DFT functionals on smaller molecules it is likely more efficient to set *bank_size = 0*, otherwise it's recommended to use the default. If a particular calculation runs out of memory, it might help to increase the number of bank processes from the default value.

The `numerically_exact` keyword will trigger algorithms that guarantee that the computed results are invariant (within double precision) with respect to the number or MPI processes. These exact algorithms require more memory and are thus not default. Even when the numbers are *not* MPI invariant they should be correct and identical within the chosen `world_prec`.

The `share_potential` keywords are used to share the memory space for the particular functions between all processes located on the same physical machine. This will save memory but it might slow the calculation down, since the shared memory cannot be "fast" memory (NUMA) for all processes at once.

Basis

This section defines the polynomial MultiWavelet basis

```
Basis {
  type = Interpolating                  # Legendre or Interpolating
  order = 7                             # Polynomial order of MW basis
}
```

The MW basis is defined by the polynomial order k, and the type of scaling functions: Legendre or Interpolating polynomials (in the current implementation it doesn't really matter which type you choose). Note that increased precision

requires higher polynomial order (use e.g $k = 5$ for $\epsilon_{rel} = 10^{-3}$, and $k = 13$ for $\epsilon_{rel} = 10^{-9}$, and interpolate in between). If the `order` keyword is left out it will be set automatically according to

$$k = -1.5 * log_{10}(\epsilon_{rel})$$

The Basis section can usually safely be omitted in the input.

Molecule

This input section specifies the geometry (given in `world_unit` units), charge and spin multiplicity of the molecule, e.g. for water (coords must be specified, otherwise defaults are shown):

```
Molecule {
  charge = 0                            # Total charge of molecule
  multiplicity = 1                      # Spin multiplicity
  translate = false                     # Translate CoM to world_origin
$coords
O   0.0000     0.0000     0.0000        # Atomic symbol and coordinate
H   0.0000     1.4375     1.1500        # Atomic symbol and coordinate
H   0.0000    -1.4375     1.1500        # Atomic symbol and coordinate
$end
}
```

Since the computational domain is always cubic and symmetric around the origin it is usually a good idea to `translate` the molecule to the origin (as long as the `world_origin` is the true origin).

WaveFunction

Here we give the wavefunction method and whether we run spin restricted (alpha and beta spins are forced to occupy the same spatial orbitals) or not (method must be specified, otherwise defaults are shown):

```
WaveFunction {
  method = <wavefunction_method>        # Core, Hartree, HF or DFT
  restricted = true                     # Spin restricted/unrestricted
}
```

There are currently four methods available: Core Hamiltonian, Hartree, Hartree-Fock (HF) and Density Functional Theory (DFT). When running DFT you can *either* set one of the default functionals in this section (e.g. `method = B3LYP`), *or* you can set `method = DFT` and specify a "non-standard" functional in the separate DFT section (see below). See *User input reference* for a list of available default functionals.

Note: Restricted open-shell wavefunctions are not supported.

DFT

This section can be omitted if you are using a default functional, see above. Here we specify the exchange-correlation functional used in DFT (functional names must be specified, otherwise defaults are shown)

```
DFT {
  spin = false                          # Use spin-polarized functionals
  density_cutoff = 0.0                  # Cutoff to set XC potential to zero
$functionals
<func1>     1.0                         # Functional name and coefficient
<func2>     1.0                         # Functional name and coefficient
$end
}
```

You can specify as many functionals as you want, and they will be added on top of each other with the given coefficient. Both exchange and correlation functionals must be set explicitly, e.g. `SLATERX` and `VWN5C` for the standard LDA functional. For hybrid functionals you must specify the amount of exact Hartree-Fock exchange as a separate functional `EXX` (`EXX 0.2` for B3LYP and `EXX 0.25` for PBE0 etc.). Option to use spin-polarized functionals or not. Unrestricted calculations will use spin-polarized functionals by default. The XC functionals are provided by the XCFun library.

Properties

Specify which properties to compute. By default, only the ground state SCF energy as well as orbital energies will be computed. Currently the following properties are available (all but the dipole moment are `false` by default)

```
Properties {
  dipole_moment = true                  # Compute dipole moment
  quadrupole_moment = false             # Compute quadrupole moment
  polarizabiltity = false               # Compute polarizability
  magnetizability = false               # Compute magnetizability
  nmr_shielding = false                 # Compute NMR shieldings
  geometric_derivative = false          # Compute geometric derivative
  plot_density = false                  # Plot converged density
  plot_orbitals = []                    # Plot converged orbitals
}
```

Some properties can be further specified in dedicated sections.

> **Warning:** The computation of the molecular gradient suffers greatly from numerical noise. The code replaces the nucleus-electron attraction with a smoothed potential. This can only partially recover the nuclear cusps, even with tight precision. The molecular gradient is only suited for use in geometry optimization of small molecules and with tight precision thresholds.

Polarizability

The polarizability can be computed with several frequencies (by default only static polarizability is computed):

```
Polarizability {
  frequency = [0.0, 0.0656]             # List of frequencies to compute
}
```

NMRShielding

For the NMR shielding we can specify a list of nuclei to compute (by default all nuclei are computed):

```
NMRShielding {
  nuclear_specific = false              # Use nuclear specific perturbation operator
  nucleus_k = [0,1,2]                   # List of nuclei to compute (-1 computes all)
}
```

The `nuclear_specific` keyword triggers response calculations using the nuclear magnetic moment operator instead of the external magnetic field. For small molecules this is not recommended since it requires a separate response calculation for each nucleus, but it might be beneficial for larger systems if you are interested only in a single shielding constant. Note that the components of the *perturbing* operator defines the *row* index in the output tensor, so `nuclear_specific = true` will result in a shielding tensor which is the transpose of the one obtained with `nuclear_specific = false`.

Plotter

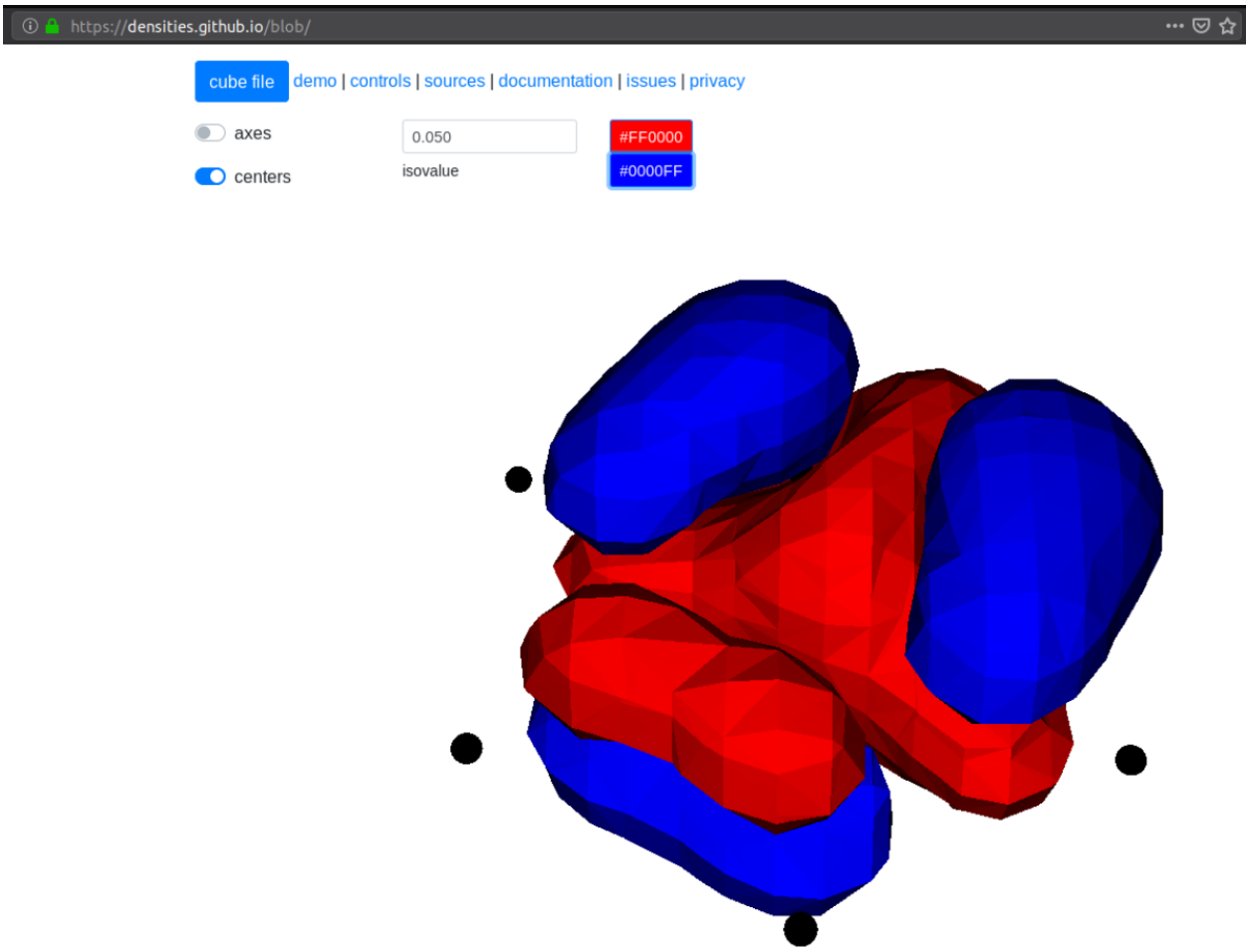
The `plot_density` and `plot_orbitals` properties will use the Plotter section to specify the parameters of the plots (by default you will get a `cube` plot on the unit cube):

```
Plotter {
  path = plots                          # File path to store plots
  type = cube                           # Plot type (line, surf, cube)
  points = [20, 20, 20]                 # Number of grid points
  O = [-4.0,-4.0,-4.0]                  # Plot origin
  A = [8.0, 0.0, 0.0]                   # Boundary vector
  B = [0.0, 8.0, 0.0]                   # Boundary vector
  C = [0.0, 0.0, 8.0]                   # Boundary vector
}
```

The plotting grid is computed from the vectors `O`, `A`, `B` and `C` in the following way:

1. `line` plot: along the vector `A` starting from `O`, using `points[0]` number of points.
2. `surf` plot: on the area spanned by the vectors `A` and `B` starting from `O`, using `points[0]` and `points[1]` points in each direction.
3. `cube` plot: on the volume spanned by the vectors `A`, `B` and `C` starting from `O`, using `points[0]`, `points[1]` and `points[2]` points in each direction.

The above example will plot on a 20x20x20 grid in the volume [-4,4]^3, and the generated files (e.g. `plots/phi_1_re.cube`) can be viewed directly in a web browser by blob , like this benzene orbital:



SCF

This section specifies the parameters for the SCF optimization of the ground state wavefunction.

SCF solver

The optimization is controlled by the following keywords (defaults shown):

```
SCF {
  run = true                          # Run SCF solver
  kain = 5                            # Length of KAIN iterative subspace
  max_iter = 100                      # Maximum number of SCF iterations
  rotation = 0                        # Iterations between diagonalize/localize
  localize = false                    # Use canonical or localized  orbitals
  start_prec = -1.0                   # Dynamic precision, start value
  final_prec = -1.0                   # Dynamic precision, final value
  orbital_thrs = 10 * world_prec      # Convergence threshold orbitals
  energy_thrs = -1.0                  # Convergence threshold energy
}
```

If `run = false` no SCF is performed, and the properties are computed directly on the initial guess wavefunction.

The `kain` (Krylov Accelerated Inexact Newton) keyword gives the length of the iterative subspace accelerator (similar to DIIS). The `rotation` keyword gives the number of iterations between every orbital rotation, which can be either localization or diagonalization, depending on the `localize` keyword. The first two iterations in the SCF are always rotated, otherwise it is controlled by the `rotation` keyword (usually this is not very important, but sometimes it fails to converge if the orbitals drift too far away from the localized/canonical forms).

The dynamic precision keywords control how the numerical precision is changed throughout the optimization. One can choose to use a lower `start_prec` in the first iterations which is gradually increased to `final_prec` (both are equal to `world_prec` by default). Note that lower initial precision might affect the convergence rate.

In general, the important convergence threshold is that of the orbitals, and by default this is set one order of magnitude higher than the overall `world_prec`. For simple energy calculations, however, it is not necessary to converge the orbitals this much due to the quadratic convergence of the energy. This means that the number of correct digits in the total energy will be saturated well before this point, and one should rather use the `energy_thrs` keyword in this case in order to save a few iterations.

Note: It is usually not feasible to converge the orbitals *beyond* the overall precision `world_prec` due to numerical noise.

Initial guess

Several types of initial guess are available:

- `core` and `sad` requires no further input and computes guesses from scratch.
- `chk` and `mw` require input files from previous MW calculations.
- `cube` requires input files computed from other sources.

The `core` and `sad` guesses are computed by diagonalizing the Hamiltonian matrix using a Core or Superposition of Atomic Densities (SAD) Hamiltonian, respectively. The matrix is constructed in a small AO basis with a given "zeta quality", which should be added as a suffix in the keyword. Available AO bases are hydrogenic orbitals of single `sz`, double `dz`, triple `tz` and quadruple `qz` zeta size.

The SAD guess can also be computed in a small GTO basis (3-21G), using the guess type `sad_gto`. In this case another input keyword `guess_screen` becomes active for screening in the MW projection of the Gaussians. The screening value is given in standard deviations. Such screening will greatly improve the efficiency of the guess for large systems. It is, however, not recommended to reduce the value much below 10 StdDevs, as this will have the *opposite* effect on efficiency due to introduction of discontinuities at the cutoff point, which leads to higher grid refinement. `sad_gto` is usually the preferred guess both for accuracy and efficiency, and is thus the default choice.

The `core` and `sad` guesses are fully specified with the following keywords (defaults shown):

```
SCF {
  guess_prec = 1.0e-3                  # Numerical precision used in guess
  guess_type = sad_gto                 # Type of inital guess (chk, mw, cube, core_XX,␣
↪sad_XX)
  guess_screen = 12.0                  # Number of StdDev before a GTO is set to zero␣
↪(sad_gto)
}
```

Checkpointing

The program can dump checkpoint files at every iteration using the `write_checkpoint` keyword (defaults shown):

```
SCF {
  path_checkpoint = checkpoint          # Path to checkpoint files
  write_checkpoint = false              # Save checkpoint files every iteration
}
```

This allows the calculation to be restarted in case it crashes e.g. due to time limit or hardware failure on a cluster. This is done by setting `guess_type = chk` in the subsequent calculation:

```
SCF {
  guess_type = chk                      # Type of inital guess (chk, mw, cube, core_XX,␣
↪sad_XX)
}
```

In this case the `path_checkpoint` must be the same as the previous calculation, as well as all other parameters in the calculation (Molecule and Basis in particular).

Write orbitals

The converged orbitals can be saved to file with the `write_orbitals` keyword (defaults shown):

```
SCF {
  path_orbitals = orbitals              # Path to orbital files
  write_orbitals = false                # Save converged orbitals to file
}
```

This will make individual files for each orbital under the `path_orbitals` directory. These orbitals can be used as starting point for subsequent calculations using the `guess_type = mw` initial guess:

```
SCF {
  guess_prec = 1.0e-3                   # Numerical precision used in guess
  guess_type = mw                       # Type of inital guess (chk, mw, cube, core_XX,␣
↪sad_XX)
}
```

Here the orbitals will be re-projected onto the current MW basis with precision `guess_prec`. We also need to specify the paths to the input files:

```
Files {
  guess_phi_p = initial_guess/phi_p     # Path to paired MW orbitals
  guess_phi_a = initial_guess/phi_a     # Path to alpha MW orbitals
  guess_phi_b = initial_guess/phi_b     # Path to beta MW orbitals
}
```

Note that by default orbitals are written to the directory called `orbitals` but the `mw` guess reads from the directory `initial_guess` (this is to avoid overwriting the files by default). So, in order to use MW orbitals from a previous calculation, you must either change one of the paths (`SCF.path_orbitals` or `Files.guess_phi_p` etc), or manually copy the files between the default locations.

Note: The `mw` guess must not be confused with the `chk` guess, although they are similar. The `chk` guess will blindly read

in the orbitals that are present, regardless of the current molecular structure and computational setup (if you run with a different computational domain or MW basis type/order the calculation will crash). The `mw` guess will re-project the old orbitals onto the new computational setup and populate the orbitals based on the *new* molecule (here the computation domain and MW basis do *not* have to match).

Response

This section specifies the parameters for the SCF optimization of the linear response functions. There might be several independent response calculations depending on the requested properties, e.g.

```
Polarizability {
  frequency = [0.0, 0.0656]             # List of frequencies to compute
}
```

will run one response for each frequency (each with three Cartesian components), while

```
Properties {
  magnetizability = true                # Compute magnetizability
  nmr_shielding = true                  # Compute NMR shieldings
}
```

will combine both properties into a single response calculation, since the perturbation operator is the same in both cases (unless you choose `NMRShielding.nuclear_specific = true`, in which case there will be a different response for each nucleus).

Response solver

The optimization is controlled by the following keywords (defaults shown):

```
Response {
  run = [true,true,true]                # Run response solver [x,y,z] direction
  kain = 5                              # Length of KAIN iterative subspace
  max_iter = 100                        # Maximum number of SCF iterations
  localize = false                      # Use canonical or localized  orbitals
  start_prec = -1.0                     # Dynamic precision, start value
  final_prec = -1.0                     # Dynamic precision, final value
  orbital_thrs = 10 * world_prec        # Convergence threshold orbitals
}
```

Each linear response calculation involves the three Cartesian components of the appropriate perturbation operator. If any of the components of `run` is `false`, no response is performed in that particular direction, and the properties are computed directly on the initial guess response functions (usually zero guess).

The `kain` (Krylov Accelerated Inexact Newton) keyword gives the length of the iterative subspace accelerator (similar to DIIS). The `localize` keyword relates to the unperturbed orbitals, and can be set independently of the `SCF.localize` keyword.

The dynamic precision keywords control how the numerical precision is changed throughout the optimization. One can choose to use a lower `start_prec` in the first iterations which is gradually increased to `final_prec` (both are equal to `world_prec` by default). Note that lower initial precision might affect the convergence rate.

For response calculations, the important convergence threshold is that of the orbitals, and by default this is set one order of magnitude higher than the overall `world_prec`.

Note: The quality of the response property depends on both the perturbed as well as the unperturbed orbitals, so they should be equally well converged.

Initial guess

The following initial guesses are available:

- `none` start from a zero guess for the response functions.
- `chk` and `mw` require input files from previous MW calculations.

By default, no initial guess is generated for the response functions, but the `chk` and `mw` guesses work similarly as for the SCF.

Checkpointing

The program can dump checkpoint files at every iteration using the `write_checkpoint` keyword (defaults shown):

```
Response {
  path_checkpoint = checkpoint          # Path to checkpoint files
  write_checkpoint = false              # Save checkpoint files every iteration
}
```

This allows the calculation to be restarted in case it crashes e.g. due to time limit or hardware failure on a cluster. This is done by setting `guess_type = chk` in the subsequent calculation:

```
Response {
  guess_type = chk                      # Type of inital guess (none, chk, mw)
}
```

In this case the `path_checkpoint` must be the same as the previous calculation, as well as all other parameters in the calculation (Molecule and Basis in particular).

Write orbitals

The converged response orbitals can be saved to file with the `write_orbitals` keyword (defaults shown):

```
Response {
  path_orbitals = orbitals              # Path to orbital files
  write_orbitals = false                # Save converged orbitals to file
}
```

This will make individual files for each orbital under the `path_orbitals` directory. These orbitals can be used as starting point for subsequent calculations using the `guess_type = mw` initial guess:

```
Response {
  guess_prec = 1.0e-3                   # Numerical precision used in guess
  guess_type = mw                       # Type of inital guess (chk, mw, cube, core_XX,␣
↪sad_XX)
}
```

Here the orbitals will be re-projected onto the current MW basis with precision `guess_prec`. We also need to specify the paths to the input files (only X for static perturbations, X and Y for dynamic perturbations):

```
Files {
  guess_X_p = initial_guess/X_p         # Path to paired MW orbitals
  guess_X_a = initial_guess/X_a         # Path to alpha MW orbitals
  guess_X_b = initial_guess/X_b         # Path to beta MW orbitals
  guess_Y_p = initial_guess/Y_p         # Path to paired MW orbitals
  guess_Y_a = initial_guess/Y_a         # Path to alpha MW orbitals
  guess_Y_b = initial_guess/Y_b         # Path to beta MW orbitals
}
```

Note that by default orbitals are written to the directory called `orbitals` but the `mw` guess reads from the directory `initial_guess` (this is to avoid overwriting the files by default). So, in order to use MW orbitals from a previous calculation, you must either change one of the paths (`Response.path_orbitals` or `Files.guess_X_p` etc), or manually copy the files between the default locations.

2.2.3 User input reference

- Keywords without a default value are **required**.
- Default values are either explicit or computed from the value of other keywords in the input.
- Sections where all keywords have a default value can be omitted.
- Predicates, if present, are the functions run to validate user input.

Keywords

world_prec
Overall relative precision in the calculation.

Type `float`

Predicates

- `1.0e-10 < value < 1.0`

world_size
Total size of computational domain given as 2**(`world_size`). Always cubic and symmetric around the origin. Negative value means it will be computed from the molecular geometry.

Type `int`

Default `-1`

Predicates

- `value <= 10`

world_unit
Length unit for *all* coordinates given in user input. Everything will be converted to atomic units (bohr) before the main executable is launched, so the JSON input is *always* given in bohrs.

Type `str`

Default `bohr`

Predicates

- `value.lower() in ["bohr", "angstrom"]`

world_origin
Global gauge origin of the calculation.

Type `List[float]`

Default `[0.0, 0.0, 0.0]`

Predicates

- `len(value) == 3`

Sections

Precisions
Define specific precision parameters.

Keywords

exchange_prec
Precision parameter used in construction of Exchange operators. Negative value means it will follow the dynamic precision in SCF.

Type `float`

Default `-1.0`

helmholtz_prec
Precision parameter used in construction of Helmholtz operators. Negative value means it will follow the dynamic precision in SCF.

Type `float`

Default `-1.0`

poisson_prec
Precision parameter used in construction of Poisson operators.

Type `float`

Default `user['world_prec']`

Predicates

- `1.0e-10 < value < 1.0`

nuclear_prec
Precision parameter used in smoothing and projection of nuclear potential.

Type `float`

Default `user['world_prec']`

Predicates

- `1.0e-10 < value < 1.0`

Printer
Define variables for printed output.

Keywords

print_level
Level of detail in the written output. Level 0 for production calculations, negative level for complete silence.

Type `int`

Default `0`

print_mpi
Write separate output from each MPI to file called `<file_name>-<mpi-rank>.out`.

Type `bool`

Default `False`

print_prec
Number of digits in property output (energies will get twice this number of digits).

Type `int`

Default `6`

Predicates

- `0 < value < 10`

print_width
Line width of printed output (in number of characters).

Type `int`

Default `75`

Predicates

- `50 < value < 100`

print_constants
Print table of physical constants used by MRChem.

Type `bool`

Default `False`

Plotter
Give details regarding the density and orbital plots. Three types of plots are available, line, surface and cube, and the plotting ranges are defined by three vectors (A, B and C) and an origin (O): `line`: plots on line spanned by A, starting from O. `surf`: plots on surface spanned by A and B, starting from O. `cube`: plots on volume spanned by A, B and C, starting from O.

Keywords

path
File path to plot directory.

Type `str`

Default `plots`

Predicates

- `value[-1] != '/'`

type
Type of plot: line (1D), surface (2D) or cube (3D).

Type `str`

Default `cube`

Predicates

- `value.lower() in ['line', 'surf', 'cube']`

points
Number of points in each direction on the cube grid.

Type `List[int]`

Default `[20, 20, 20]`

Predicates

- `all(p > 0 for p in value)`
- `not (user['Plotter']['type'] == 'line' and len(value) < 1)`
- `not (user['Plotter']['type'] == 'surf' and len(value) < 2)`
- `not (user['Plotter']['type'] == 'cube' and len(value) < 3)`

O
Origin of plotting ranges.

Type `List[float]`

Default `[0.0, 0.0, 0.0]`

Predicates

- `len(value) == 3`

A
First boundary vector for plot.

Type `List[float]`

Default `[1.0, 0.0, 0.0]`

Predicates

- `len(value) == 3`

B
Second boundary vector for plot.

Type `List[float]`

Default `[0.0, 1.0, 0.0]`

Predicates

- `len(value) == 3`

C
Third boundary vector for plot.

Type `List[float]`

Default `[0.0, 0.0, 1.0]`

Predicates

- `len(value) == 3`

MPI
Define MPI related parameters.

Keywords

numerically_exact
This will use MPI algorithms that guarantees that the output is invariant wrt the number of MPI processes.

Type `bool`

Default `False`

shared_memory_size
Size (MB) of the MPI shared memory blocks of each shared function.

Type `int`

Default `10000`

share_nuclear_potential
This will use MPI shared memory for the nuclear potential.

Type `bool`

Default `False`

share_coulomb_potential
This will use MPI shared memory for the Coulomb potential.

Type `bool`

Default `False`

share_xc_potential
This will use MPI shared memory for the exchange-correlation potential.

Type `bool`

Default `False`

bank_size
Number of MPI processes exclusively dedicated to manage orbital bank.

Type `int`

Default `-1`

Basis
Define polynomial basis.

Keywords

order
Polynomial order of multiwavelet basis. Negative value means it will be set automatically based on the world precision.

Type `int`

Default `-1`

type
Polynomial type of multiwavelet basis.

Type `str`

Default `interpolating`

Predicates

- `value.lower() in ['interpolating', 'legendre']`

Derivatives
Define various derivative operators used in the code.

Keywords

kinetic
Derivative used in kinetic operator.

Type `str`

Default `abgv_55`

h_b_dip
Derivative used in magnetic dipole operator.

Type `str`

Default `abgv_00`

h_m_pso
Derivative used in paramagnetic spin-orbit operator.

Type `str`

Default `abgv_00`

Molecule
Define molecule.

Keywords

charge
Total charge of molecule.

Type `int`

Default `0`

multiplicity
Spin multiplicity of molecule.

Type `int`

Default `1`

Predicates

- `value > 0`

translate
Translate coordinates such that center of mass coincides with the global gauge origin.

Type `bool`

Default `False`

coords
Coordinates in xyz format. Atoms can be given either using atom symbol or atom number

Type `str`

WaveFunction
Define the wavefunction method.

Keywords

method
Wavefunction method. See predicates for valid methods. `hf`, `hartreefock` and `hartree-fock` all mean the same thing, while `lda` is an alias for `svwn5`. You can set a non-standard DFT functional (e.g. varying the amount of exact exchange) by choosing `dft` and specifing the functional(s) in the `DFT` section below.

Type `str`

Predicates

- `value.lower() in ['core', 'hartree', 'hf', 'hartreefock', 'hartree-fock', 'dft', 'lda', 'svwn3', 'svwn5', 'pbe', 'pbe0', 'bpw91', 'bp86', 'b3p86', 'b3p86-g', 'blyp', 'b3lyp', 'b3lyp-g', 'olyp', 'kt1', 'kt2', 'kt3']`

restricted
Use spin restricted wavefunction.

Type `bool`

Default `True`

environment
Set method for treatment of environment. `none` for vacuum calculation. `PCM` for Polarizable Continuum Model, which will activate the `PCM` input section for further parametrization options.

Type `str`

Default `none`

Predicates

- `value.lower() in ['none', 'pcm']`

DFT
Define the exchange-correlation functional in case of DFT.

Keywords

density_cutoff
Hard cutoff for passing density values to XCFun.

Type `float`

Default `0.0`

functionals
List of density functionals with numerical coefficient. E.g. for PBE0 `EXX 0.25`, `PBEX 0.75`, `PBEC 1.0`, see XCFun documentation <https://xcfun.readthedocs.io/>_.

Type `str`

Default `` ``

spin
Use spin separated density functionals.

Type `bool`

Default `not(user['WaveFunction']['restricted'])`

Properties
Provide a list of properties to compute (total SCF energy and orbital energies are always computed).

Keywords

dipole_moment
Compute dipole moment.

Type `bool`

Default `True`

quadrupole_moment
Compute quadrupole moment. Note: Gauge origin dependent, should be used with `translate = true` in Molecule.

Type `bool`

Default `False`

polarizability
Compute polarizability tensor.

Type `bool`

Default `False`

magnetizability
Compute magnetizability tensor.

Type `bool`

Default `False`

nmr_shielding
Compute NMR shielding tensor.

Type `bool`

Default `False`

geometric_derivative
Compute geometric derivative.

Type `bool`

Default `False`

plot_density
Plot converged electron density.

Type `bool`

Default `False`

plot_orbitals
Plot converged molecular orbitals from list of indices, negative index plots all orbitals.

Type `List[int]`

Default `[]`

ExternalFields
Define external electromagnetic fields.

Keywords

electric_field
Strength of external electric field.

Type `List[float]`

Default `[]`

Predicates

- `len(value) == 0 or len(value) == 3`

Polarizability
Give details regarding the polarizability calculation.

Keywords

frequency
List of external field frequencies.

Type `List[float]`

Default `[0.0]`

NMRShielding
Give details regarding the NMR shileding calculation.

Keywords

nuclear_specific
Use nuclear specific perturbation operator (h_m_pso).

Type `bool`

Default `False`

nucleus_k
List of nuclei to compute. Negative value computes all nuclei.

Type `List[int]`

Default `[-1]`

Files
Defines file paths used for program input/output. Note: all paths must be given in quotes if they contain slashes "path/to/file".

Keywords

guess_basis
File name for GTO basis set, used with `gto` guess.

Type `str`

Default `initial_guess/mrchem.bas`

guess_gto_p
File name for paired orbitals, used with `gto` guess.

Type `str`

Default `initial_guess/mrchem.mop`

guess_gto_a
File name for alpha orbitals, used with `gto` guess.

Type `str`

Default `initial_guess/mrchem.moa`

guess_gto_b
File name for beta orbitals, used with `gto` guess.

Type `str`

Default `initial_guess/mrchem.mob`

guess_phi_p
File name for paired orbitals, used with `mw` guess. Expected path is ``<path_orbitals>/phi_p_scf_idx_<0…Np>_<re/im>.mw

Type `str`

Default `initial_guess/phi_p`

guess_phi_a
File name for alpha orbitals, used with `mw` guess. Expected path is ``<path_orbitals>/phi_a_scf_idx_<0…Na>_<re/im>.mw

Type `str`

Default `initial_guess/phi_a`

guess_phi_b
File name for beta orbitals, used with `mw` guess. Expected path is ``<path_orbitals>/phi_b_scf_idx_<0…Nb>_<re/im>.mw

Type `str`

Default `initial_guess/phi_b`

guess_x_p
File name for paired response orbitals, used with `mw` guess. Expected path is ``<path_orbitals>/x_p_rsp_idx_<0…Np>_<re/im>.mw

Type `str`

Default `initial_guess/X_p`

guess_x_a
File name for alpha response orbitals, used with `mw` guess. Expected path is ``<path_orbitals>/x_a_rsp_idx_<0…Na>_<re/im>.mw

Type `str`

Default `initial_guess/X_a`

guess_x_b
File name for beta response orbitals, used with `mw` guess. Expected path is ``<path_orbitals>/x_b_rsp_idx_<0…Nb>_<re/im>.mw

Type `str`

Default `initial_guess/X_b`

guess_y_p
File name for paired response orbitals, used with `mw` guess. Expected path is ``<path_orbitals>/y_p_rsp_idx_<0…Np>_<re/im>.mw

Type `str`

Default `initial_guess/Y_p`

guess_y_a
File name for alpha response orbitals, used with `mw` guess. Expected path is ``<path_orbitals>/y_a_rsp_idx_<0…Na>_<re/im>.mw

Type `str`

Default `initial_guess/Y_a`

guess_y_b
File name for beta response orbitals, used with `mw` guess. Expected path is ``<path_orbitals>/y_b_rsp_idx_<0…Nb>_<re/im>.mw

Type `str`

Default `initial_guess/Y_b`

guess_cube_p
File name for paired orbitals, used with `cube` guess. Expected path is ``<path_orbitals>/phi_p_scf_idx_<0…Np>_<re/im>.cube

Type `str`

Default `initial_guess/phi_p`

guess_cube_a
File name for alpha orbitals, used with `cube` guess. Expected path is ``<path_orbitals>/phi_a>_scf_idx_<0…Na>_<re/im>.cube

Type `str`

Default `initial_guess/phi_a`

guess_cube_b
File name for beta orbitals, used with `cube` guess. Expected path is ``<path_orbitals>/phi_b_scf_idx_<0…Nb>_<re/im>.cube

Type `str`

Default `initial_guess/phi_b`

cube_vectors
Directory where cube vectors are stored for mrchem calculation.

Type `str`

Default `cube_vectors/`

SCF
Includes parameters related to the ground state SCF orbital optimization.

Keywords

run
Run SCF solver. Otherwise properties are computed on the initial orbitals.

Type `bool`

Default `True`

max_iter
Maximum number of SCF iterations.

Type `int`

Default `100`

kain
Length of KAIN iterative history.

Type `int`

Default `5`

rotation
Number of iterations between each diagonalization/localization.

Type `int`

Default `0`

localize
Use canonical or localized orbitals.

Type `bool`

Default `False`

energy_thrs
Convergence threshold for SCF energy.

Type `float`

Default `-1.0`

guess_prec
Precision parameter used in construction of initial guess.

Type `float`

Default `0.001`

Predicates

- `1.0e-10 < value < 1.0`

guess_screen
Screening parameter used in GTO evaluations, in number of standard deviations. Every coordinate beyond N StdDev from the Gaussian center is evaluated to zero. Note that too aggressive screening is counter productive, because it leads to a sharp cutoff in the resulting function which requires higher grid refinement. Negative value means no screening.

Type `float`

Default `12.0`

start_prec
Incremental precision in SCF iterations, initial value.

Type `float`

Default `-1.0`

final_prec
Incremental precision in SCF iterations, final value.

Type `float`

Default `-1.0`

guess_type
Type of initial guess for ground state orbitals. `chk` restarts a previous calculation which was dumped using the `write_checkpoint` keyword. This

will load MRA and electron spin configuration directly from the checkpoint files, which are thus required to be identical in the two calculations. `mw` will start from final orbitals in a previous calculation written using the `write_orbitals` keyword. The orbitals will be re-projected into the new computational setup, which means that the electron spin configuration and MRA can be different in the two calculations. `gto` reads precomputed GTO orbitals (requires extra non-standard input files for basis set and MO coefficients). `core` and `sad` will diagonalize the Fock matrix in the given AO basis (SZ, DZ, TZ or QZ) using a Core or Superposition of Atomic Densities Hamiltonian, respectively.

Type `str`

Default `sad_dz`

Predicates

- `value.lower() in ['mw', 'chk', 'gto', 'core_sz', 'core_dz', 'core_tz', 'core_qz', 'sad_sz', 'sad_dz', 'sad_tz', 'sad_qz', 'sad_gto', 'cube']`

write_checkpoint
Write orbitals to disk in each iteration, file name <`path_checkpoint`>/`phi_scf_idx_`<`0..N`>. Can be used as `chk` initial guess in subsequent calculations. Note: must be given in quotes if there are slashes in the path "path/to/checkpoint".

Type `bool`

Default `False`

path_checkpoint
Path to checkpoint files during SCF, used with `write_checkpoint` and `chk` guess.

Type `str`

Default `checkpoint`

Predicates

- `value[-1] != '/'`

write_orbitals
Write final orbitals to disk, file name <`path_orbitals`>/`phi_`<`p/a/b`>`_scf_idx_`<`0..Np/Na/Nb`>. Can be used as `mw` initial guess in subsequent calculations.

Type `bool`

Default `False`

path_orbitals
Path to where converged orbitals will be written in connection with the `write_orbitals` keyword. Note: must be given in quotes if there are slashes in the path "path/to/orbitals".

Type `str`

Default `orbitals`

Predicates

- `value[-1] != '/'`

orbital_thrs
Convergence threshold for orbital residuals.

Type `float`

Default `10 * user['world_prec']`

Response
Includes parameters related to the response SCF optimization.

Keywords

run
In which Cartesian directions to run response solver.

Type `List[bool]`

Default `[True, True, True]`

max_iter
Maximum number of response iterations.

Type `int`

Default `100`

kain
Length of KAIN iterative history.

Type `int`

Default `5`

property_thrs
Convergence threshold for symmetric property. Symmetric meaning the property computed from the same operator as the response purturbation, e.g. for external magnetic field the symmetric property corresponds to the magnetizability (NMR shielding in non-symmetric, since one of the operators is external magnetic field, while the other is nuclear magnetic moment).

Type `float`

Default `-1.0`

start_prec
Incremental precision in SCF iterations, initial value.

Type `float`

Default `-1.0`

final_prec
Incremental precision in SCF iterations, final value.

Type `float`

Default `-1.0`

guess_prec
Precision parameter used in construction of initial guess.

Type `float`

Default `0.001`

Predicates

- 1.0e-10 < value < 1.0

guess_type
Type of initial guess for response. none will start from a zero guess for the response functions. chk restarts a previous calculation which was dumped using the write_checkpoint keyword. mw will start from final orbitals in a previous calculation written using the write_orbitals keyword. The orbitals will be re-projected into the new computational setup.

Type str

Default none

Predicates

- value.lower() in ['none', 'chk', 'mw']

write_checkpoint
Write perturbed orbitals to disk in each iteration, file name <path_checkpoint>/<X/Y>_rsp_<direction>_idx_<0..N>. Can be used as chk initial guess in subsequent calculations.

Type bool

Default False

path_checkpoint
Path to checkpoint files during SCF, used with write_checkpoint and chk guess.

Type str

Default checkpoint

Predicates

- value[-1] != '/'

write_orbitals
Write final perturbed orbitals to disk, file name <path_orbitals>/<X/Y>_<p/a/b>_rsp_<direction>_idx_<0..Np/Na/Nb>. Can be used as mw initial guess in subsequent calculations.

Type bool

Default False

path_orbitals
Path to where converged orbitals will be written in connection with the write_orbitals keyword.

Type str

Default orbitals

Predicates

- value[-1] != '/'

orbital_thrs
Convergence threshold for orbital residuals.

Type float

Default 10 * user['world_prec']

localize
Use canonical or localized unperturbed orbitals.

Type bool

Default user['SCF']['localize']

PCM
Includes parameters related to the computation of the reaction field energy of a system in an environment within the Polarizable Continuum Model.

Sections

SCRF
Parameters for the Self-Consistent Reaction Field optimization.

Keywords

max_iter
Max number of iterations allowed in the nested procedure.

Type int

Default 100

dynamic_thrs
Set the convergence threshold for the nested procedure. true will dynamically tighten the convergence threshold based on the absolute value of the latest orbital update as. When the orbitals are close to convergence (mo_residual < world_prec*10) the convergence threshold will be set equal to world_prec. false uses world_prec as convergence threshold throughout.

Type bool

Default True

optimizer
Choose which function to use in the KAIN solver, the surface charge density (gamma) or the reaction potential (V_R).

Type str

Default potential

Predicates

- value.lower() in ['density', 'potential']

density_type
What part of the total molecular charge density to use in the algorithm. total uses the total charge density. nuclear uses only the nuclear part of the total charge density. electronic uses only the electronic part of the total charge density.

Type str

Default total

Predicates

- value.lower() in ['total', 'nuclear', 'electronic']

kain
Number of previous reaction field iterates kept for convergence acceleration during the nested precedure.

Type `int`

Default `user['SCF']['kain']`

Cavity
Define the interlocking spheres cavity.

Keywords

mode
Determines how to set up the interlocking spheres cavity. `atoms`: centers are taken from the molecular geometry, radii taken from tabulated data (van der Waals radius), and rescaled using the parameters `alpha`, `beta` and `sigma` (R_i <- alpha*R_i + beta*sigma). Default spheres can be modified and/or extra spheres added, using the *$spheres* section, see documentation. `explicit`: centers and radii given explicitly in the `spheres` block.

Type `str`

Default `atoms`

Predicates

- `value.lower() in ['atoms', 'explicit']`

spheres
This input parameter affects the list of spheres used to generate the cavity. In all cases, values for the radius, the radius scaling factor (`alpha`), the width (`sigma`), and the width scaling factor (`beta`) can be modified. If they are not specified their global default values are used. In `atoms` mode, we *modify* the default list of spheres, built with centers from the molecular geometry and radii from internal tabulated van der Waals values. To *substitute* a sphere, include a line like: $spheres i R [alpha] [beta] [sigma] $end to specify that the `i` atom in the molecule (0-based indexing) should use radius `R` instead of the pre-tabulated vdW radius. To *add* a sphere, include a line like: $spheres x y z R [alpha] [beta] [sigma] $end to specify that a sphere of radius `R` should be added at position `(x, y, z)`. Spheres added in this way are not aware of their parent atom, if any. They will **not** contribute to the molecular gradient. In `explicit` mode, we *build* the complete sphere list from scratch. You can add a line like: $spheres x y z R [alpha] [beta] [sigma] $end to specify that a sphere of radius `R` should be added at position `(x, y, z)`. Spheres added in this way are not aware of their parent atom, if any. They will **not** contribute to the molecular gradient. Alternatively, you can specify a line like: $spheres i R [alpha] [beta] [sigma] $end to specify that the `i` atom in the molecule (0-based indexing) should use radius `R`. Spheres added in this way are aware of their parent atom. They will contribute to the molecular gradient.

Type `str`

Default ````

alpha
Scaling factor on the radius term for the cavity rescaling (R_i <- alpha*R_i + beta*sigma). Only used for the default vdW radii in *atoms* mode, not if explicit `$spheres` are given.

Type `float`

Default `1.1`

beta
Scaling factor on the boundary width term for the cavity rescaling (R_i <- alpha*R_i + beta*sigma). Only used for the default vdW radii in *atoms* mode, not if explicit `$spheres` are given.

Type `float`

Default `0.5`

sigma
Width of cavity boundary, smaller value means sharper transition.

Type `float`

Default `0.2`

Permittivity
Parameters for the permittivity function.

Keywords

epsilon_in
Permittivity inside the cavity. 1.0 is the permittivity of free space, anything other than this is undefined behaviour.

Type `float`

Default `1.0`

epsilon_out
Permittivity outside the cavity. This is characteristic of the solvent used.

Type `float`

Default `1.0`

formulation
Formulation of the Permittivity function. Currently only the exponential is used.

Type `str`

Default `exponential`

Predicates

- `value.lower() in ['exponential']`

Constants

Physical and mathematical constants used by MRChem

Keywords

hartree2simagnetizability

Conversion factor for magnetizability from atomic units to SI units (unit: J T^-2). Affected code: Printed value of the magnetizability property.

Type `float`

Default `78.9451185`

light_speed

Speed of light in atomic units (unit: au). Affected code: Relativistic Hamiltonians (ZORA, etc.)

Type `float`

Default `137.035999084`

angstrom2bohrs

Conversion factor for Cartesian coordinates from Angstrom to Bohr (unit: Å^-1). Affected code: Parsing of input coordinates, printed coordinates

Type `float`

Default `1.8897261246257702`

hartree2kjmol

Conversion factor from Hartree to kJ/mol (unit: kJ mol^-1). Affected code: Printed value of energies.

Type `float`

Default `2625.4996394798254`

hartree2kcalmol

Conversion factor from Hartree to kcal/mol (unit: kcal mol^-1). Affected code: Printed value of energies.

Type `float`

Default `627.5094740630558`

hartree2ev

Conversion factor from Hartree to eV (unit: ev). Affected code: Printed value of energies.

Type `float`

Default `27.211386245988`

hartree2wavenumbers

Conversion factor from Hartree to wavenumbers (unit: cm^-1). Affected code: Printed value of frequencies.

Type `float`

Default `219474.6313632`

fine_structure_constant

Fine-structure constant in atomic units (unit: au). Affected code: Certain magnetic interaction operators.

Type `float`

Default `0.0072973525693`

electron_g_factor

Electron g factor in atomic units (unit: au). Affected code: Certain magnetic interaction operators.

Type `float`

Default `-2.00231930436256`

dipmom_au2debye

Conversion factor for dipoles from atomic units to Debye (unit: ?). Affected code: Printed value of dipole moments.

Type `float`

Default `2.5417464739297717`

2.2.4 Running MRChem with QCEngine

MRChem >=1.0 can be used as a computational engine with the QCEngine program executor. QCEngine can be useful for running calculations on large sets of molecules and input parameters. The results are collected in standardised QCScheme format, which makes it easy to build post-processing pipelines and store data according to Findability, Accessibility, Interoperability, and Reuse (FAIR) of digital assets principles. Furthermore, QCEngine provides different geometry optimization drivers that can use the molecular gradient computed by MRChem for structural optimization.

Installation

The easiest way is to install both QCEngine and MRChem in a Conda environment using the precompiled version:

```
conda create -n mrchem-qcng mrchem qcengine qcelemental geometric optking pip -c conda-
↪forge
conda activate mrchem-qcng
python -m pip install -U pyberny
```

It is also possible to use your own installation of MRChem: just make sure that the installation folder is in your `PATH`.

Note: If you want to use the precompiled, MPI-parallel version of MRChem with OpenMPI, install `mrchem=*=*openmpi*` insted of just `mrchem`. A binary package compiled against MPICH is also available: `mrchem=*=*mpich*`.

Single compute

Calculations in QCEngine are defined in Python scripts. For example, the following runs MRChem to obtain the energy of water:

```
import qcelemental as qcel
import qcengine as qcng


mol = qcel.models.Molecule(geometry=[[0, 0, 0], [0, 1.5, 0], [0, 0, 1.5]],
                           symbols=["O", "H", "H"],
                           connectivity=[[0, 1, 1], [0, 2, 1]])
print(mol)

computation = {
    "molecule": mol,
    "driver": "energy",
    "model": {"method": "HF"},
    "keywords": {"world_prec": 1.0e-3},
}
ret = qcng.compute(computation, "mrchem")

print(f"E_HF = {ret.return_result} Hartree")
```

You can save this sample as *mrchem-run-hf.py* and execute it with:

```
python mrchem-run-hf.py
```

Which will print to screen:

```
Molecule(name='H2O', formula='H2O', hash='b41d0c5')
E_HF = -75.9789291596064 Hartree
```

Note that:

1. The molecule is specified, in Angstrom, using a QCElemental object.
2. The computation is described using a Python dictionary.
3. The `driver` selects the kind of calculation you want to run with MRChem. Available drivers are:
    - `energy`, for single-point energy calculations.
    - `gradient`, for evaluation of the molecular gradient at a given geometry.
    - `properties`, for the calculation of molecular properties.
4. The `model` selects the wavefunction: `HF` for Hartree-Fock and any of the DFT functionals known to MRChem for a corresponding DFT calculation.
5. The `keywords` key in the dictionary accepts a dictionary of MRChem options. Any of the options in the usual input file are recognized.

Once you have a dictionary defining your computation, you can run it with:

```
ret = qcng.compute(computation, "mrchem")
```

You can reuse the same dictionary with *multiple* computational engine, *e.g.* other quantum chemistry programs that are recognized as executors by QCEngine. The return value from the `compute` function contains all data produced

during the calculation in QCSchema format including, for example, the execution time elapsed. The full JSON output produced by MRChem is also available and can be inspected in Python as:

```
mrchem_json_out = ret.extras["raw_output"]["output"]
```

The full, human-readable input is saved as the `stdout` property of the object returned by `compute`.

Parallelism

QCEngine allows you to exploit available parallel hardware. For example, to use 20 OpenMP threads in your MRChem calculation you would provide an additional task configuration dictionary as a `task_config` argument to `compute`:

```
ret = qcng.compute(
        computation,
        "mrchem",
        task_config={"ncores": 20})
```

You can inspect how the job was launched by printing out the `provenance` dictionary:

```
print(ret.extras["raw_output"]["output"]["provenance"])
```

```
{
 "creator": "MRChem",
 "mpi_processes": 1,
 "routine": "/home/roberto/miniconda3/envs/mrchem-qcng/bin/mrchem.x",
 "total_cores": 1,
 "version": "1.1.0",
 "ncores": 12,
 "nnodes": 1,
 "ranks_per_node": 1,
 "cores_per_rank": 12,
 "total_ranks": 1
}
```

It is also possible to run MPI-parallel and hybrid MPI+OpenMP jobs. Assuming that you installed the MPICH version of the MRChem MPI-parallel Conda package, the basic `task_config` argument to `compute` would look like:

```
task = {
  "nnodes": 1,  # number of nodes
  "ncores": 12,  # number of cores per task on each node
  "cores_per_rank": 6,  # number of cores per MPI rank
  "use_mpiexec": True,  # launch with MPI
  "mpiexec_command": "mpiexec -n {total_ranks}",  # the invocation of MPI
}
```

This task configuration will launch a MPI job with 2 ranks on a single node. Each rank has access to 6 cores for OpenMP parallelization. The `provenance` dictionary now shows:

```
{
 "creator": "MRChem",
 "mpi_processes": 2,
 "routine": "mpiexec -n 2 /home/roberto/miniconda3/envs/mrchem-qcng/bin/mrchem.x",
 "total_cores": 12,
```

(continues on next page)

(continued from previous page)

```
 "version": "1.1.0",
 "ncores": 12,
 "nnodes": 1,
 "ranks_per_node": 2,
 "cores_per_rank": 6,
 "total_ranks": 2
}
```

The `mpiexec_command` is a string that will be interpolated to provide the exact invocation. In the above example, MRChem will be run with:

```
mpiexec -n 2 /home/roberto/miniconda3/envs/mrchem-qcng/bin/mrchem.x
```

The following interpolation parameters are understood by QCEngine when creating the MPI invocation:

- `{nnodes}`: number of nodes.
- `{cores_per_rank}`: number of cores to use for each MPI rank.
- `{ranks_per_node}`: number of MPI ranks per node. Computed as `ncores // cores_per_rank`.
- `{total_ranks}`: total number of MPI ranks. Computed as `nnodes * ranks_per_node`.

More complex MPI invocations are possible by setting the appropriate `mpiexec_command` in the task configuration. For usage with a scheduler, such as SLURM, you should refer to the documentation of your computing cluster and the documentation of QCEngine.

Geometry optimizations

Running geometry optimizations is just as easy as single compute. The following example optimizes the structure of water using the SVWN5 functional with MW4. The geomeTRIC package is used as optimization driver, but pyberny or optking would also work.

Warning: The computation of the molecular gradient can be affected by significant numerical noise for MW3 and MW4, to the point that it can be impossible to converge a geometry optimization. Using a tighter precision might help, but the cost of the calculation might be prohibitively large.

```
import qcelemental as qcel
import qcengine as qcng

mol =  qcel.models.Molecule(
    geometry=[
        [ 0.29127930, 3.00875625, 0.20308515],
        [-1.21253048, 1.95820900, 0.10303324],
        [ 0.10002049, 4.24958115,-1.10222079]
    ],
    symbols=["O", "H", "H"],
    fix_com=True,
    fix_orientation=True,
    fix_symmetry="c1")

opt_input =  {
```

(continues on next page)

(continued from previous page)

```
    "keywords": {
        "program": "mrchem",
        "maxiter": 70
    },
    "input_specification": {
        "driver": "gradient",
        "model": {
            "method": "SVWN5",
        },
        "keywords": {
            "world_prec": 1.0e-4,
            "SCF": {
                "guess_type": "core_dz",
            }
        }
    },
    "initial_molecule": mol,
}

opt = qcng.compute_procedure(
        opt_input,
        "geometric",
        task_config={"ncores": 20})

print(opt.stdout)

print("==> Optimized geometry <==")
print(opt.final_molecule.pretty_print())

print("==> Optimized geometric parameters <==")
for m in [[0, 1], [0, 2], [1, 0, 2]]:
    opt_val = opt.final_molecule.measure(m)
    print(f"Internal degree of freedom {m} = {opt_val:.3f}")
```

Running this script will print all the steps taken during the structural optimization. The final printout contains the optimized geometry:

```
Geometry (in Angstrom), charge = 0.0, multiplicity = 1:

   Center              X                  Y                  Z
------------   -----------------  -----------------  -----------------
O                -4.146209038013     2.134923126314    -3.559202294678
H                -4.906566693905     1.536801624016    -3.587431156799
H                -4.270830051398     2.773072094238    -4.275607223691
```

and the optimized values of bond distances and bond angle:

```
Internal degree of freedom [0, 1] = 1.829
Internal degree of freedom [0, 2] = 1.828
Internal degree of freedom [1, 0, 2] = 106.549
```

2.2.5 Program input/output file

Input schema

```
"input": {
  "schema_name": string,                    # Name of the input schema
  "schema_version": int,                    # Version of the input schema
  "molecule": {                             # Section for Molecule specification
    "charge": int,                          # Total molecular charge
    "multiplicity": int,                    # Total spin multiplicity
    "coords": array[                        # Array of atoms
      {                                     # (one entry per atom)
        "atom": string,                     # Atomic symbol
        "xyz": array[float]                 # Nuclear Cartesian coordinate
      }
    ],
    "cavity": {
      "spheres": array[                     # Array of cavity spheres
        {                                   # (one entry per sphere)
          "center": array[float],           # Cartesian coordinate of sphere center
          "radius": float                   # Radius of cavity sphere
          "alpha": float                    # Scaling factor of radius
          "beta": float                     # Scaling factor of width
          "sigma": float                    # Width of cavity boundary
        }
      ],
    }
  },
  "mpi": {                                  # Section for MPI specification
    "bank_size": int,                       # Number of MPI ranks in memory bank
    "numerically_exact": bool,              # Guarantee MPI invariant results
    "shared_memory_size": int               # Size (MB) of MPI shared memory blocks
  },
  "mra": {                                  # Section for MultiResolution Analysis
    "basis_type": string,                   # Basis type (interpolating/legendre)
    "basis_order": int,                     # Polynomial order of basis
    "max_scale": int,                       # Maximum level of refinement
    "min_scale": int,                       # Minimum level of refinement (root scale)
    "boxes": array[int],                    # Number of root boxes
    "corner": array[int]                    # Translation of first root box
  },
  "printer": {                              # Section for printed output
    "file_name": string,                    # Name of output file
    "print_level": int,                     # Amount of printed output
    "print_mpi": bool,                      # Use separate output file for each MPI
    "print_prec": int,                      # Number of digits for printed output
    "print_width": int                      # Line width of printed output
  },
  "scf_calculation": {                      # Section for SCF specification
    "fock_operator": {                      # Contributions to Fock operator
      "kinetic_operator": {                 # Add Kinetic operator to Fock
        "derivative": string                # Type of derivative operator
      },
```

(continues on next page)

(continued from previous page)

```
      "nuclear_operator": {                  # Add Nuclear operator to Fock
        "proj_prec": float,                  # Projection prec for potential
        "smooth_prec": float,                # Smoothing parameter for potential
        "shared_memory": bool                # Use shared memory for potential
      },
      "coulomb_operator": {                  # Add Coulomb operator to Fock
        "poisson_prec": float,               # Build prec for Poisson operator
        "shared_memory": bool                # Use shared memory for potential
      },
      "exchange_operator": {                 # Add Exchange operator to Fock
        "poisson_prec": float,               # Build prec for Poisson operator
        "screen": bool                       # Use screening in Exchange operator
      },
      "reaction_operator": {                 # Add Reaction operator to Fock
        "poisson_prec": float,               # Precision for Poisson operator
        "kain": int,                         # Length of KAIN history in nested SCRF␣
↪procedure
        "max_iter": int,                     # Maximum number of iterations in nested␣
↪SCRF procedure
        "optimizer": string,                 # Use density or potential in KAIN solver
        "dynamic_thrs": bool,                # Use static or dynamic convergence␣
↪threshold
        "density_type": string,              # Type of charge density [total, nuclear,␣
↪electronic]
        "epsilon_in": float,                 # Permittivity inside the cavity
        "epsilon_out": float,                # Permittivity outside the cavity
        "formulation": string                # Formulation of the permittivity function
      },
      "xc_operator": {                       # Add XC operator to Fock
        "shared_memory": bool,               # Use shared memory for potential
        "xc_functional": {                   # XC functional specification
          "spin": bool,                      # Use spin separated functional
          "cutoff": float,                   # Cutoff value for small densities
          "functionals": array[              # Array of density functionals
            {
              "coef": float,                 # Numerical coefficient
              "name": string                 # Functional name
            }
          ]
        }
      },
      "external_operator": {                 # Add external field operator to Fock
        "electric_field": array[float],      # Electric field vector
        "r_O": array[float]                  # Gauge orgigin for electric field
      }
    },
    "initial_guess": {                       # Initial guess specification
      "type": string,                        # Type of initial guess
      "prec": float,                         # Precision for initial guess
      "zeta": int,                           # Zeta quality for AO basis
      "method": string,                      # Name of method for initial energy
      "localize": bool,                      # Use localized orbitals
```

(continues on next page)

(continued from previous page)

```
    "restricted": bool,                 # Use spin restricted orbitals
    "relativity": string,               # Name of relativistic method
    "screen": float,                    # Screening used in GTO evaluations
    "file_chk": string,                 # Path to checkpoint file
    "file_basis": string,               # Path to GTO basis file
    "file_gto_a": string,               # Path to GTO MO file (alpha)
    "file_gto_b": string,               # Path to GTO MO file (beta)
    "file_gto_p": string,               # Path to GTO MO file (paired)
    "file_phi_a": string,               # Path to MW orbital file (alpha)
    "file_phi_b": string,               # Path to MW orbital file (beta)
    "file_phi_p": string,               # Path to MW orbital file (paired)
    "file_CUBE_a": str,                 # Path to CUBE orbital file (alpha)
    "file_CUBE_b": str,                 # Path to CUBE orbital file (beta)
    "file_CUBE_p": str                  # Path to CUBE orbital file (paired)
  },
  "scf_solver": {                       # SCF solver specification
    "kain": int,                        # Length of KAIN history
    "max_iter": int,                    # Maximum number of iterations
    "method": string,                   # Name of electronic structure method
    "relativity": string,               # Name of relativistic method
    "rotation": int,                    # Iterations between localize/diagonalize
    "localize": bool,                   # Use localized orbitals
    "checkpoint": bool,                 # Save checkpoint file
    "file_chk": string,                 # Name of checkpoint file
    "start_prec": float,                # Start precision for solver
    "final_prec": float,                # Final precision for solver
    "helmholtz_prec": float,            # Precision for Helmholtz operators
    "orbital_thrs": float,              # Convergence threshold orbitals
    "energy_thrs":float                 # Convergence threshold energy
  },
  "properties": {                       # Collection of properties to compute
    "dipole_moment": {                  # Collection of dipole moments
      id (string): {                    # Unique id: 'dip-${number}'
        "precision": float,             # Operator precision
        "operator": string,             # Operator used for property
        "r_O": array[float]             # Operator gauge origin
      }
    },
    "quadrupole_moment": {              # Collection of quadrupole moments
      id (string): {                    # Unique id: 'quad-${number}'
        "precision": float,             # Operator precision
        "operator": string,             # Operator used for property
        "r_O": array[float]             # Operator gauge origin
      }
    },
    "geometric_derivative": {           # Collection of geometric derivatives
      id (string): {                    # Unique id: 'geom-${number}'
        "precision": float,             # Operator precision
        "operator": string,             # Operator used for property
        "smooth_prec": float            # Smoothing parameter for potential
      }
    }
```

(continues on next page)

(continued from previous page)

```
    },
    "plots": {                                  # Collection of plots to perform
      "density": bool,                          # Plot converged densities
      "orbitals": array[int],                   # List of orbitals to plot
      "plotter": {                              # Section specifying plotting parameters
        "path": string,                         # Path to output files
        "type": string,                         # Type of plot (line, surf or cube)
        "points": array[int],                   # Number of points in each direction
        "O": array[float],                      # Plotting range origin
        "A": array[float],                      # Plotting range A vector
        "B": array[float],                      # Plotting range B vector
        "C": array[float]                       # Plotting range C vector
    }
  },
  "rsp_calculations": {                         # Collection of response calculations
    id (string): {                              # Response id: e.g. 'ext_el-${frequency}'
      "dynamic": bool,                          # Use dynamic response solver
      "frequency": float,                       # Perturbation frequency
      "perturbation": {                         # Perturbation operator
        "operator": string                      # Operator used in response calculation
      },
      "components": array[                      # Array of perturbation components
        {                                       # (one per Cartesian direction)
          "initial_guess": {                    # Initial guess specification
            "type": string,                     # Type of initial guess
            "prec": float,                      # Precision for initial guess
            "file_chk_x": string,               # Path to checkpoint file for X
            "file_chk_y": string,               # Path to checkpoint file for Y
            "file_x_a": string,                 # Path to MW file for X (alpha)
            "file_x_b": string,                 # Path to MW file for X (beta)
            "file_x_p": string,                 # Path to MW file for X (paired)
            "file_y_a": string,                 # Path to MW file for Y (alpha)
            "file_y_b": string,                 # Path to MW file for Y (beta)
            "file_y_p": string                  # Path to MW file for Y (paired)
          },
          "rsp_solver": {                       # Response solver specification
            "kain": int,                        # Length of KAIN history
            "max_iter": int,                    # Maximum number of iterations
            "method": string,                   # Name of electronic structure method
            "checkpoint": bool,                 # Save checkpoint file
            "file_chk_x": string,               # Name of X checkpoint file
            "file_chk_y": string,               # Name of Y checkpoint file
            "orth_prec": float,                 # Precision for orthogonalization
            "start_prec": float,                # Start precision for solver
            "final_prec": float,                # Final precision for solver
            "helmholtz_prec": float,            # Precision for Helmholtz operators
            "orbital_thrs": float,              # Convergence threshold orbitals
            "property_thrs": float              # Convergence threshold property
          }
        }
      ],
      "properties": {                           # Collection of properties to compute
```

(continues on next page)

(continued from previous page)

```
    "polarizability": {                 # Collection of polarizabilities
      id (string): {                    # Unique id: 'pol-${frequency}'
        "precision": float,             # Operator precision
        "operator": string,             # Operator used for property
        "r_O": array[float]             # Operator gauge origin
      }
    },
    "magnetizability": {                # Collection of magnetizabilities
      id (string): {                    # Unique id: 'mag-${frequency}'
        "frequency": float,             # Perturbation frequency
        "precision": float,             # Operator precision
        "dia_operator": string,         # Operator used for diamagnetic property
        "para_operator": string,        # Operator used for paramagnetic property
        "derivative": string,           # Operator derivative type
        "r_O": array[float]             # Operator gauge origin
      }
    },
    "nmr_shielding": {                  # Collection of NMR shieldings
      id (string): {                    # Unique id: 'nmr-${nuc_idx}${atom_symbol}'
        "precision": float,             # Operator precision
        "dia_operator": string,         # Operator used for diamagnetic property
        "para_operator": string,        # Operator used for paramagnetic property
        "derivative": string,           # Operator derivative type
        "smoothing": float,             # Operator smoothing parameter
        "r_O": array[float],            # Operator gauge origin
        "r_K": array[float]             # Nuclear coordinate
      }
    }
  },
  "fock_operator": {                    # Contributions to perturbed Fock operator
    "coulomb_operator": {               # Add Coulomb operator to Fock
      "poisson_prec": float,            # Build prec for Poisson operator
      "shared_memory": bool             # Use shared memory for potential
    },
    "exchange_operator": {              # Add Exchange operator to Fock
      "poisson_prec": float,            # Build prec for Poisson operator
      "screen": bool                    # Use screening in Exchange operator
    },
    "xc_operator": {                    # Add XC operator to Fock
      "shared_memory": bool,            # Use shared memory for potential
      "xc_functional": {                # XC functional specification
        "spin": bool,                   # Use spin separated functional
        "cutoff": float,                # Cutoff value for small densities
        "functionals": array[           # Array of density functionals
          {
            "coef": float,              # Numerical coefficient
            "name": string              # Functional name
          }
        ]
      }
    }
  },
```

(continues on next page)

(continued from previous page)

```
    "unperturbed": {                          # Section for unperturbed part of response
      "prec": float,                          # Precision used for unperturbed system
      "localize": bool,                       # Use localized unperturbed orbitals
      "fock_operator": {                      # Contributions to unperturbed Fock operator
        "kinetic_operator": {                 # Add Kinetic operator to Fock
          "derivative": string                # Type of derivative operator
        },
        "nuclear_operator": {                 # Add Nuclear operator to Fock
          "proj_prec": float,                 # Projection prec for potential
          "smooth_prec": float,               # Smoothing parameter for potential
          "shared_memory": bool               # Use shared memory for potential
        },
        "coulomb_operator": {                 # Add Coulomb operator to Fock
          "poisson_prec": float,              # Build prec for Poisson operator
          "shared_memory": bool               # Use shared memory for potential
        },
        "exchange_operator": {                # Add Exchange operator to Fock
          "poisson_prec": float,              # Build prec for Poisson operator
          "screen": bool                      # Use screening in Exchange operator
        },
        "xc_operator": {                      # Add XC operator to Fock
          "shared_memory": bool,              # Use shared memory for potential
          "xc_functional": {                  # XC functional specification
            "spin": bool,                     # Use spin separated functional
            "cutoff": float,                  # Cutoff value for small densities
            "functionals": array[             # Array of density functionals
              {
                "coef": float,                # Numerical coefficient
                "name": string                # Functional name
              }
            ]
          }
        },
        "external_operator": {                # Add external field operator to Fock
          "electric_field": array[float],     # Electric field vector
          "r_O": array[float]                 # Gauge orgigin for electric field
        }
      }
    }
  },
  "constants": {                              # Physical constants used throughout MRChem
    "angstrom2bohrs": float,                  # Conversion factor from Angstrom to Bohr
    "dipmom_au2debye": float,                 # Conversion factor from atomic units to␣
↪Debye
    "electron_g_factor": float,               # Electron g factor in atomic units
    "fine_structure_constant": float,         # Fine-structure constant in atomic units
    "hartree2ev": float,                      # Conversion factor from Hartree to eV
    "hartree2kcalmol": float,                 # Conversion factor from Hartree to kcal/mol
    "hartree2kjmol": float,                   # Conversion factor from Hartree to kJ/mol
    "hartree2simagnetizability": float,       # Conversion factor from Hartree to J T^-2
    "hartree2wavenumbers": float,             # Conversion factor from Hartree to cm^-1
```

(continues on next page)

(continued from previous page)

```
    "light_speed": float                    # Speed of light in vacuo in atomic units
  }
}
```

Output schema

```
"output": {
  "success": bool,                          # Whether all requested calculations␣
↪succeeded
  "schema_name": string,                    # Name of the output schema
  "schema_version": int,                    # Version of the output schema
  "provenance": {                           # Information on how the results were␣
↪obtained
    "creator": string,                      # Program name
    "version": string,                      # Program version
    "nthreads": int,                        # Number of OpenMP threads used
    "mpi_processes": int,                   # Number of MPI processes used
    "total_cores": int,                     # Total number of cores used
    "routine": string                       # The function that generated the output
  },
  "properties": {                           # Collection of final properties
    "charge": int,                          # Total molecular charge
    "multiplicity": int,                    # Total spin multiplicity
    "center_of_mass": array[float],         # Center of mass coordinate
    "geometry": array[                      # Array of atoms
      {                                     # (one entry per atom)
        "symbol": string,                   # Atomic symbol
        "xyz": array[float]                 # Cartesian coordinate
      }
    ],
    "orbital_energies": {                   # Collection of orbital energies
      "spin": array[string],                # Array of spins ('p', 'a' or 'b')
      "energy": array[float],               # Array of energies
      "occupation": array[int],             # Array of orbital occupations
      "sum_occupied": float                 # \sum_i occupation[i]*energy[i]
    },
    "scf_energy": {                         # Collection of energy contributions
      "E_kin": float,                       # Kinetic energy
      "E_nn": float,                        # Classical nuclear-nuclear interaction
      "E_en": float,                        # Classical electron-nuclear interaction
      "E_ee": float,                        # Classical electron-electron interaction
      "E_next": float,                      # Classical nuclear-external field␣
↪interaction
      "E_eext": float,                      # Classical electron-external field␣
↪interaction
      "E_x": float,                         # Hartree-Fock exact exchange energy
      "E_xc": float,                        # DFT exchange-correlation energy
      "E_el": float,                        # Sum of electronic contributions
      "E_nuc": float,                       # Sum of nuclear contributions
      "E_tot": float,                       # Sum of all contributions
```

(continues on next page)

(continued from previous page)

```
    "Er_el": float,                          # Electronic reaction energy
    "Er_nuc": float,                         # Nuclear reaction energy
    "Er_tot": float                          # Sum of all reaction energy contributions
  },
  "dipole_moment": {                         # Collection of electric dipole moments
    id (string): {                           # Unique id: 'dip-${number}'
      "r_O": array[float],                   # Gauge origin vector
      "vector": array[float],                # Total dipole vector
      "vector_el": array[float],             # Electronic dipole vector
      "vector_nuc": array[float],            # Nuclear dipole vector
      "magnitude": float                     # Magnitude of total vector
    }
  },
  "quadrupole_moment": {                     # Collection of electric quadrupole moments
    id (string): {                           # Unique id: 'quad-${number}'
      "r_O": array[float],                   # Gauge origin vector
      "tensor": array[float],                # Total quadrupole tensor
      "tensor_el": array[float],             # Electronic quadrupole tensor
      "tensor_nuc": array[float]             # Nuclear quadrupole tensor
    }
  },
  "polarizability": {                        # Collection of polarizabilities
    id (string): {                           # Unique id: 'pol-${frequency}'
      "frequency": float,                    # Perturbation frequency
      "r_O": array[float],                   # Gauge origin vector
      "tensor": array[float],                # Full polarizability tensor
      "isotropic_average": float             # Diagonal average
    }
  },
  "magnetizability": {                       # Collection of magnetizability
    id (string): {                           # Unique id: 'mag-${frequency}'
      "frequency": float,                    # Perturbation frequency
      "r_O": array[float],                   # Gauge origin vector
      "tensor": array[float],                # Full magnetizability tensor
      "tensor_dia": array[float],            # Diamagnetic tensor
      "tensor_para": array[float],           # Paramagnetic tensor
      "isotropic_average": float             # Diagonal average
    }
  },
  "nmr_shielding": {                         # Collection of NMR shielding tensors
    id (string): {                           # Unique id: 'nmr-${nuc_idx}-${atom_symbol}'
      "r_O": array[float],                   # Gauge origin vector
      "r_K": array[float],                   # Nuclear coordinate vector
      "tensor": array[float],                # Full NMR shielding tensor
      "tensor_dia": array[float],            # Diamagnetic tensor
      "tensor_para": array[float],           # Paramagnetic tensor
      "diagonalized_tensor": array[float], # Diagonalized tensor used for (an)isotropy
      "isotropic_average": float,            # Diagonal average
      "anisotropy": float                    # Anisotropy of tensor
    }
  },
  "geometric_derivative": {                  # Collection of geometric derivatives
```

(continues on next page)

(continued from previous page)

```
      id (string): {                          # Unique id: 'geom-${number}'
        "electronic": array[float],           # Electronic component of the geometric␣
↪derivative
        "electronic_norm": float,             # Norm of the electronic component of the␣
↪geoemtric derivative
        "nuclear": array[float],              # Nuclear component of the geometric␣
↪derivative
        "nuclear_norm": float,                # Norm of the nuclear component of the␣
↪geometric derivative
        "total": array[float],                # Geometric derivative
        "total_norm": float                   # Norm of the geometric derivative
      }
    }
  },
  "scf_calculation": {                        # Ground state SCF calculation
    "success": bool,                          # SCF finished successfully
    "initial_energy": {                       # Energy computed from initial orbitals
      "E_kin": float,                         # Kinetic energy
      "E_nn": float,                          # Classical nuclear-nuclear interaction
      "E_en": float,                          # Classical electron-nuclear interaction
      "E_ee": float,                          # Classical electron-electron interaction
      "E_next": float,                        # Classical nuclear-external field␣
↪interaction
      "E_eext": float,                        # Classical electron-external field␣
↪interaction
      "E_x": float,                           # Hartree-Fock exact exchange energy
      "E_xc": float,                          # DFT exchange-correlation energy
      "E_el": float,                          # Sum of electronic contributions
      "E_nuc": float,                         # Sum of nuclear contributions
      "E_tot": float,                         # Sum of all contributions
      "Er_el": float,                         # Electronic reaction energy
      "Er_nuc": float,                        # Nuclear reaction energy
      "Er_tot": float                         # Sum of all reaction energy contributions
    },
    "scf_solver": {                           # Details from SCF optimization
      "converged": bool,                      # Optimization converged
      "wall_time": float,                     # Wall time (sec) for SCF optimization
      "cycles": array[                        # Array of SCF cycles
        {                                     # (one entry per cycle)
          "energy_total": float,              # Current total energy
          "energy_update": float,             # Current energy update
          "mo_residual": float,               # Current orbital residual
          "wall_time": float,                 # Wall time (sec) for SCF cycle
          "energy_terms": {                   # Energy contributions
            "E_kin": float,                   # Kinetic energy
            "E_nn": float,                    # Classical nuclear-nuclear interaction
            "E_en": float,                    # Classical electron-nuclear interaction
            "E_ee": float,                    # Classical electron-electron interaction
            "E_next": float,                  # Classical nuclear-external field␣
↪interaction
            "E_eext": float,                  # Classical electron-external field␣
↪interaction
```

(continues on next page)

(continued from previous page)

```
            "E_x": float,                    # Hartree-Fock exact exchange energy
            "E_xc": float,                   # DFT exchange-correlation energy
            "E_el": float,                   # Sum of electronic contributions
            "E_nuc": float,                  # Sum of nuclear contributions
            "E_tot": float,                  # Sum of all contributions
            "Er_el": float,                  # Electronic reaction energy
            "Er_nuc": float,                 # Nuclear reaction energy
            "Er_tot": float                  # Sum of all reaction energy contributions
          }
        }
      ]
    }
  },
  "rsp_calculations": {                      # Collection of response calculations
    id (string): {                           # Response id: e.g. 'ext_el-${frequency}'
      "success": bool,                       # Response finished successfully
      "frequency": float,                    # Frequency of perturbation
      "perturbation": string,                # Name of perturbation operator
      "components": array[                   # Array of operator components
        {                                    # (one entry per Cartesian direction)
          "rsp_solver": {                    # Details from response optimization
            "wall_time": float,              # Wall time (sec) for response calculation
            "converged": bool,               # Optimization converged
            "cycles": array[                 # Array of response cycles
              {                              # (one entry per cycle)
                "symmetric_property": float, # Property computed from perturbation␣
↪operator
                "property_update": float,    # Current symmetric property update
                "mo_residual": float,        # Current orbital residual
                "wall_time": float           # Wall time (sec) for response cycle
              }
            ]
          }
        }
      ]
    }
  }
}
```

2.3 Programmer's Manual

2.3.1 Classes and functions reference

Chemistry

Classes for the chemistry overlay

Environment

Classes for the solvent environment overlay

Cavity

class **Cavity** : public mrcpp::RepresentableFunction<3>

Interlocking spheres cavity centered on the nuclei of the molecule. The *Cavity* class represents the following function Fosso-Tande2013.

$$C(\mathbf{r}) = 1 - \prod_{i=1}^{N}(1 - C_i(\mathbf{r}))$$
$$C_i(\mathbf{r}) = 1 - \frac{1}{2}\left(1 + \text{erf}\left(\frac{|\mathbf{r} - \mathbf{r}_i| - R_i}{\sigma_i}\right)\right)$$

where $\mathbf{r}$ is the coordinate of a point in 3D space, $\mathbf{r}_i$ is the coordinate of the i-th nucleus, R_i is the radius of the i-th sphere, and σ_i is the width of the transition between the inside and outside of the cavity. The transition has a sigmoidal shape, such that the boundary is a smooth function instead of sharp boundaries often seen in other continuum models. This function is 1 inside and 0 outside the cavity.

The radii are computed as:

$$R_i = \alpha_i R_{0,i} + \beta_i \sigma_i$$

where:

- $R_{0,i}$ is the atomic radius. By default, the van der Waals radius.
- α_i is a scaling factor. By default, 1.1
- β_i is a width scaling factor. By default, 0.5
- σ_i is the width. By default, 0.2

Public Functions

Cavity(const std::vector<mrcpp::Coord<3>> &coords, const std::vector<double> &R, const std::vector<double> &alphas, const std::vector<double> &betas, const std::vector<double> &sigmas)

Initializes the members of the class and constructs the analytical gradient vector of the *Cavity*.

Cavity(const std::vector<mrcpp::Coord<3>> &coords, const std::vector<double> &R, double sigma)

Initializes the members of the class and constructs the analytical gradient vector of the *Cavity*.

This CTOR applies a single width factor to the cavity and **does** not modify the radii. That is, in the formula:

$$R_i = \alpha_i R_{0,i} + \beta_i \sigma_i$$

for every atom i, $\alpha_i = 1.0$ and $\beta_i = 0.0$.

double **evalf**(const mrcpp::Coord<3> &r) const override

Evaluates the value of the cavity at a 3D point **r**.

Parameters

r – coordinate of 3D point at which the *Cavity* is to be evaluated at.

Returns

double value of the *Cavity* at point **r**

inline std::vector<mrcpp::Coord<3>> **getCoordinates**() const

Returns *centers*.

inline std::vector<double> **getOriginalRadii**() const

Returns *radii_0*.

inline std::vector<double> **getRadii**() const

Returns *radii*.

inline std::vector<double> **getRadiiScalings**() const

Returns *alphas*.

inline std::vector<double> **getWidths**() const

Returns *sigmas*.

inline std::vector<double> **getWidthScalings**() const

Returns *betas*.

Protected Attributes

std::vector<double> **radii_0**

Contains the *unscaled* radius of each sphere in #Center.

std::vector<double> **alphas**

The radius scaling factor for each sphere.

std::vector<double> **betas**

The width scaling factor for each sphere.

std::vector<double> **sigmas**

The width for each sphere.

std::vector<double> **radii**

Contains the radius of each sphere in #Center. $R_i = \alpha_i R_{0,i} + \beta_i \sigma_i$.

std::vector<mrcpp::Coord<3>> **centers**

Contains each of the spheres centered on the nuclei of the Molecule.

Related

auto **gradCavity**(const mrcpp::Coord<3> &r, int index, const std::vector<mrcpp::Coord<3>> &centers, const std::vector<double> &radii, const std::vector<double> &widths) -> double

Constructs a single element of the gradient of the *Cavity*.

This constructs the analytical partial derivative of the *Cavity* C with respect to x, y or z coordinates and evaluates it at a point $\mathbf{r}$. This is given for x by

$$\frac{\partial C(\mathbf{r})}{\partial x} = (1 - C(\mathbf{r})) \sum_{i=1}^{N} - \frac{(x - x_i)\, e^{-\frac{s_i^2(\mathbf{r})}{\sigma^2}}}{\sqrt{\pi}\sigma \left(0.5 \operatorname{erf}\left(\frac{s_i(\mathbf{r})}{\sigma}\right) + 0.5\right) |\mathbf{r} - \mathbf{r}_i|}$$

where the subscript i is the index related to each sphere in the cavity, and s is the signed normal distance from the surface of each sphere.

Parameters

- **r** – The coordinates of a test point in 3D space.
- **index** – An integer that defines the variable of differentiation (0->x, 1->z and 2->z).
- **centers** – A vector containing the coordinates of the centers of the spheres in the cavity.
- **radii** – A vector containing the radii of the spheres.
- **width** – A double value describing the width of the transition at the boundary of the spheres.

Returns

A double number which represents the value of the differential (w.r.t. x, y or z) at point r.

Permittivity

class **Permittivity** : public mrcpp::RepresentableFunction<3>

Permittivity function related to a substrate molecule and a solvent continuum. The *Permittivity* class represents the following function Fosso-Tande2013.

$$\epsilon(\mathbf{r}) = \epsilon_{in} \exp\left(\left(\log \frac{\epsilon_{out}}{\epsilon_{in}}\right)(1 - C(\mathbf{r}))\right)$$

where $\mathbf{r}$ is the coordinate of a point in 3D space, C is the *cavity* function of the substrate, and ϵ_{in} and ϵ_{out} are the dielectric constants describing, respectively, the permittivity *inside* and *outside* the *cavity* of the substrate.

Public Functions

Permittivity(const *Cavity* cavity, double epsilon_in, double epsilon_out, std::string formulation)

Standard constructor. Initializes the *cavity*, *epsilon_in* and *epsilon_out* with the input parameters.

Parameters

- **cavity** – interlocking spheres of *Cavity* class.
- **epsilon_in** – permittivity inside the *cavity*.
- **epsilon_out** – permittivity outside the *cavity*.
- **formulation** – Decides which formulation of the *Permittivity* function to implement, only exponential available as of now.

double **evalf**(const mrcpp::Coord<3> &r) const override

Evaluates *Permittivity* at a point in 3D space with respect to the state of *inverse*.

Parameters

r – coordinates of a 3D point in space.

Returns

$\frac{1}{\epsilon(\mathbf{r})}$ if *inverse* is true, and $\epsilon(\mathbf{r})$ if *inverse* is false.

inline void **flipFunction**(bool is_inverse)

Changes the value of *inverse*.

inline auto **isInverse**() const

Returns the current state of *inverse*.

inline auto **getCoordinates**() const

Calls the *Cavity::getCoordinates()* method of the *cavity* instance.

inline auto **getRadii**() const

Calls the *Cavity::getRadii()* method of the *cavity* instance.

inline auto **getGradVector**() const

Calls the Cavity::getGradVector() method of the *cavity* instance.

inline auto **getEpsIn**() const

Returns the value of *epsilon_in*.

inline auto **getEpsOut**() const

Returns the value of *epsilon_out*.

inline *Cavity* **getCavity**() const

Returns the cavity.

inline std::string **getFormulation**() const

Returns the formulation.

void **printParameters**() const

Print parameters.

Private Members

bool **inverse** = false

State of *evalf* .

double **epsilon_in**

Dielectric constant describing the permittivity of free space.

double **epsilon_out**

Dielectric constant describing the permittivity of the solvent.

std::string **formulation**

Formulation of the permittivity function, only exponential is used as of now.

Cavity **cavity**

A *Cavity* class instance.

SCRF

class **SCRF**

class that performs the computation of the *ReactionPotential*, named Self Consistent Reaction Field.

Private Members

mrcpp::FunctionTreeVector<3> **d_cavity**

Vector containing the 3 partial derivatives of the cavity function.

Initial Guess

Classes providing the initial guess of the orbitals

Properties

Classes for the calculation of molecular properties

Quantum Mechanical Functions

Classes to handle quantum mechanical functions such as electronic density, molecular orbitals.

QMOperators

The classes that implement quantum mechanical operators

QMPotential

class **QMPotential**

Operator defining a multiplicative potential.

Inherits the general features of a complex function from QMFunction and implements the multiplication of this function with an Orbital. The actual function representing the operator needs to be implemented in the derived classes, where the *re and *im FunctionTree pointers should be assigned in the setup() function and deallocated in the clear() function.

XCOperator

class **XCOperator**

DFT Exchange-Correlation operator containing a single *XCPotential*.

This class is a simple TensorOperator realization of

XCPotential

class **XCPotential**

Exchange-Correlation potential defined by a particular (spin) density.

The XC potential is computed by mapping of the density through a XC functional, provided by the XCFun library. There are two ways of defining the density:

1) Use getDensity() prior to setup() and build the density as you like. 2) Provide a default set of orbitals in the constructor that is used to compute the density on-the-fly in setup().

If a set of orbitals has NOT been given in the constructor, the density MUST be explicitly computed prior to setup(). The density will be computed on-the-fly in setup() ONLY if it is not already available. After setup() the operator will be fixed until clear(), which deletes both the density and the potential.

LDA and GGA functionals are supported as well as two different ways to compute the XC potentials: either with explicit derivatives or gamma-type derivatives.

ReactionPotential

class **ReactionPotential** : public mrchem::QMPotential

class containing the solvent-substrate interaction reaction potential obtained by solving

$$\Delta V_R = -4\pi \left(\rho \frac{1-\epsilon}{\epsilon} + \gamma_s \right)$$

where ρ is the total molecular density of a solute molecule, ϵ is the *Permittivity* function of the continuum and γ_s is the surface charge distribution.

Public Functions

ReactionPotential(std::unique_ptr<*SCRF*> scrf_p, std::shared_ptr<mrchem::OrbitalVector> Phi_p)

Initializes the *ReactionPotential* class.

Parameters

- **scrf_p** – A *SCRF* instance which contains the parameters needed to compute the *ReactionPotential*.
- **Phi_p** – A pointer to a vector which contains the orbitals optimized in the SCF procedure.

inline void **updateMOResidual**(double const err_t)

Updates the helper.mo_residual member variable. This variable is used to set the convergence criterion in the dynamic convergence method.

Private Members

std::unique_ptr<*SCRF*> **helper**

A *SCRF* instance used to compute the *ReactionPotential*.

std::shared_ptr<mrchem::OrbitalVector> **Phi**

holds the Orbitals needed to compute the electronic density for the *SCRF* procedure.

SCF Solver

Classes for the resolution of the SCF equations of HF and DFT

INDEX

M

Q

X