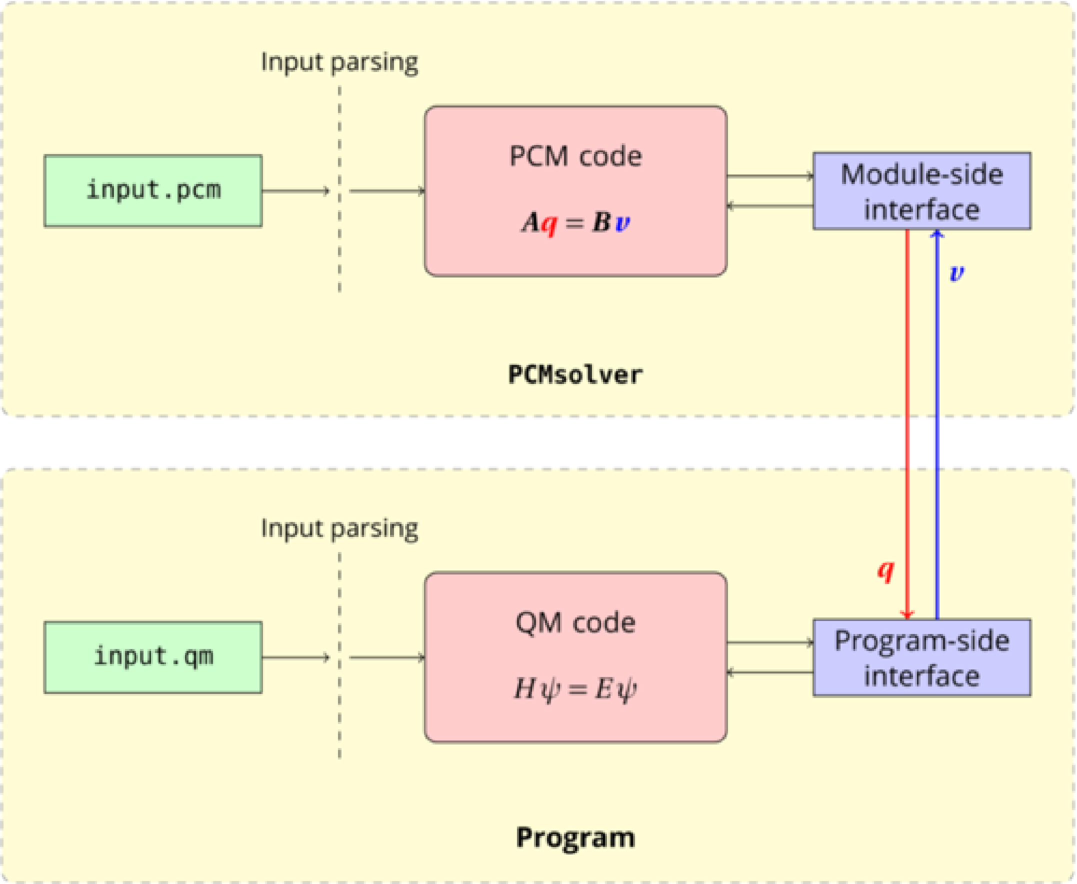
PCMSolver

Roberto Di Remigio, Luca Frediani and contributors

Dec 01, 2020

TABLE OF CONTENTS

This is the documentation for the PCMSolver application programming interface. PCMSolver is an API for solving the Polarizable Continuum Model electrostatic problem [TMC05]



With PCMSolver we aim to:

1. provide a plug-and-play library for adding the PCM functionality to *any* quantum chemistry program;
2. create a playground for easily extending the implementation of the model.

PCMSolver is distributed under the terms of the GNU Lesser General Public License. An archive with the currently released source can be found on GitHub.

```
@misc{PCMSolver,
  note = "{\texttt{PCMSolver}, an open-source library for the polarizable continuum␣
↪model
  electrostatic problem, written by R.~Di~Remigio,
  L.~Frediani and contributors (see http://pcmsolver.readthedocs.io/)}"
  doi= "10.5281/zenodo.1156166"
}
```

PCMSolver has been added to the following quantum chemistry programs

- Psi4
- DALTON
- LSDALTON
- DIRAC

- ReSpect
- KOALA

Don’t see you code listed here? Please contact us.

CHAPTER

ONE

PCMSOLVER USERS' MANUAL

1.1 Building the module

PCMSolver configuration and build process is managed through CMake.

1.1.1 Prerequisites and dependencies

A number of prerequisites and dependencies are to be satisfied to successfully build the module. It will be here assumed that you want to perform a "full" build, i.e. you want to build the static libraries to be linked to your QM program, the unit test suite and an offline copy of this documentation.

Compilers

- a C++ compiler, compliant with the 2011 ISO C++ standard. The build system will downgrade to using the 1998 ISO C++ standard plus the 2003 technical corrigendum and some additional defect reports, if no suitable support if found.

  > **Warning:** Backwards compatibility support for the C++03 standard is **deprecated** and will be removed in upcoming releases of the library.

- a C compiler, compliant with the ISO C99 standard.
- a Fortran compiler, compliant with the Fortran 2003 standard.

The list of primary test environments can be found in the README.md file. It is entirely possible that using other compiler versions you might be able to build the module. In order to ensure that you have a sane build, you will have to run the unit test suite.

Libraries and toolchain programs

- CMake version 3.3 and higher;
- Git version 1.7.1 and higher;
- Python interpreter 2.7 and higher;
- Boost libraries version 1.54.0 and higher;

Note: Version 1.54.0 of Boost libraries is shipped with the module and resides in the `cmake/downloaded` subdirectory. Unless you want to use another version of Boost, you should not worry about satisfying this dependency.

- zlib version 1.2 and higher (unit test suite only);
- Doxygen version 1.7.6 and higher (documentation only)
- Perl (documentation only)
- Sphinx (documentation only)

PCMSolver relies on the Eigen template libraries version 3.3.0 and higher. Version 3.3.0 of Eigen libraries is shipped with the module and resides in the `external` subdirectory.

1.1.2 Configuration

Configuration is managed through the front-end script `setup.py` residing in the repository main directory. Issuing:

```
./setup [options] [build path]
```

will create the build directory in build path and run CMake with the given options. By default, files are configured in the `build` directory. The `-h` or `--help` option will list the available options and their effect. Options can be forwarded directly to CMake by using the `--cmake-options` flag and listing the `-D...` options. Usually the following command is sufficient to get the configuration done for a debug build, including compilation of the unit test suite:

```
./setup --type=debug
```

The unit tests suite is **always** compiled in standalone mode, unless the `-DENABLE_TESTS=OFF` option is forwarded to CMake.

Getting Boost

You can get Boost libraries in two ways:

- already packaged by your Linux distribution or through MacPorts/Brew;
- by downloading the archive from http://www.boost.org/ and building it yourself.

In case your distribution packages a version older than 1.54.0 you might chose to either build Boost on your own or to rely on the automated build of the necessary Boost libraries when compiling the module (recommended). Full documentation on how to build Boost on Unix variants is available here. It is here assumed that the user **does not** have root access to the machine and will install the libraries to a local prefix, a subdirectory of `/home/user-name` tipically. Once you've downloaded and unpacked the archive, run the bootstrap script to configure:

```
cd path/to/boost
./bootstrap.sh --prefix=/home/user-name/boost
```

Running `./bootstrap.sh --help` will list the available options for the script. To build run:

```
./b2 install
```

This might take a while. After a successful build you will find the headers in `/home/user-name/boost/include` and libraries in `/home/user-name/boost/lib` Now, you will have Boost in a nonstandard location. Without hints CMake will not be able to find it and configuration of *PCMSolver* will fail. To avoid this, you will have to pass the location of the headers and libraries to the setup script, either with:

```
./setup --boost-headers=/home/user-name/boost/include --boost-libs=/home/user-name/
↪boost/lib
```

or with:

```
./setup -DBOOST_INCLUDEDIR=/home/user-name/boost/include -DBOOST_LIBRARYDIR=/home/
↪user-name/boost/lib
```

Advanced configuration options

These options are marked as advanced as it is highly unlikely they will be useful when not programming the library:

- `--exdiag` Enable C++ extended diagnostics flags. Disabled by default.
- `--ccache` Enable use of ccache for C/C++ compilation caching. Enabled by default, unless ccache is not available.
- `--build-boost` Deactivate Boost detection and build on-the-fly. Disabled by default.
- `--eigen` Root directory for Eigen3. Search for Eigen3 in the location provided by the user. If search fails, fall back to the version bundled with the library.
- `--static` Create only static library. Disabled by default.

Some options can only be tweaked *via* `--cmake-options` to the setup script:

- `ENABLE_DOCS` Enable build of documentation. This requires a number of additional dependencies. If any of these are not met, documentation is not built. Enabled by default.
- `ENABLE_LOGGER` Enable compilation of logger sources. Disabled by default.

  Warning: The logger is not currently in use in any part of the code.

- `ENABLE_TIMER` Enable compilation of timer sources. Enabled by default.
- `BUILD_STANDALONE` Enable compilation of standalone `run_pcm` executable. Enabled by default.
- `TEST_Fortran_API` Test the Fortran 90 bindings for the API. Enabled by default.
- `ENABLE_GENERIC` Enable mostly static linking in shared library. Disabled by default.
- `ENABLE_TESTS` Enable compilation of unit tests suite. Enabled by default.
- `SHARED_LIBRARY_ONLY` Create only shared library. Opposite of `--static`.
- `PYMOD_INSTALL_LIBDIR` *If set*, installs python scripts/modules to `${CMAKE_INSTALL_LIBDIR}${PYMOD_INSTALL_LIBDIR}/pcmsolver` rather than the default `${CMAKE_INSTALL_BINDIR}` (i.e., `bin`).
- `CMAKE_INSTALL_BINDIR` Where to install executables, if not to `bin`.
- `CMAKE_INSTALL_LIBDIR` Where to install executables, if not to `bin`.
- `CMAKE_INSTALL_INCLUDESDIR` Where to install executables, if not to `bin`.
- `CMAKE_INSTALL_BINDIR` Location within `CMAKE_INSTALL_PREFIX` (`--prefix`) to which executables are installed (default: `bin`).
- `CMAKE_INSTALL_LIBDIR` Location within `CMAKE_INSTALL_PREFIX` (`--prefix`) to which libraries are installed (default: `lib`).

- `CMAKE_INSTALL_INCLUDEDIR` Location within `CMAKE_INSTALL_PREFIX` (`--prefix``) to which headers are installed (default: `include`).
- `PYMOD_INSTALL_LIBDIR` *If set*, location within `CMAKE_INSTALL_LIBDIR` to which python modules are installed, `${CMAKE_INSTALL_LIBDIR}/${PYMOD_INSTALL_LIBDIR}/pcmsolver`. *If not set*, python modules installed to default `${CMAKE_INSTALL_LIBDIR}/python/pcmsolver`.

1.1.3 Build and test

To compile and link, just go to the build directory and run:

```
make -j N
```

where `N` is the number of cores you want to use when building.

Note: Building on more than one core can sometimes result in a "race condition" and a crash. If that happens, please report the problem as an issue on our issue tracker on GitHub. Running `make` on a single core might get you through compilation.

To run the whole test suite:

```
ctest -j N
```

You can also use CTest to run a specific test or a set of tests. For example:

```
ctest -R gepol
```

will run all the test containing the string "gepol" in their name.

1.2 Input description

PCMSolver needs a number of input parameters at runtime. The API provides two ways of providing them:

1. by means of an additional input file, parsed by the `go_pcm.py` script;
2. by means of a special section in the host program input.

Method 1 is more flexible: all parameters that can be modified by the user are available. The host program needs only copy the additional input file to the scratch directory before execution. Method 2 just gives access to the core parameters.

In this page, input style and input parameters available in Method 1 will be documented.

Note that it is also possible to run the module standalone and use a classical charge distribution. The classical charge distribution can be specified by giving a molecular geometry in the molecule section and an additional point multipoles distribution in the charge distribution section. The `run_pcm` executable has to be compiled for a standalone run with:

```
python <build-path/bin>/go_pcm.py --exe <build-path/bin> --inp molecule.inp
```

where the `molecule.inp` input file looks like:

```
units = angstrom
codata = 2002
medium
```

(continues on next page)

(continued from previous page)

```
{
        solvertype = cpcm
        correction = 0.5
    solvent = cyclohexane
}

cavity
{
        type = gepol
        area = 0.6
        radiiset = uff
        mode = implicit
}

molecule
{
    # x, y, z, q
    geometry = [0.000000000, 0.00000000,  0.08729478, 9.0,
                0.000000000, 0.00000000, -1.64558444, 1.0]
}
```

The script and the executable do not need to be in the same directory.

1.2.1 Input style

The input for PCMSolver is parsed through the Getkw library written by Jonas Juselius and is organized in **sections** and **keywords**. Input reading is case-insensitive. An example input structure is shown below, there are also some working examples in the directory `examples`. A general input parameter has the following form (Keyword = [Data type]):

```
Units = [String]
CODATA = [Integer]
Cavity {
       Type = [String]
       NpzFile = [String]
       Area = [Double]
       Scaling = [Bool]
       RadiiSet = [String]
       MinRadius = [Double]
       Mode = [String]
       Atoms = [Array of Integers]
       Radii = [Array of Doubles]
       Spheres = [Array of Doubles]
}
Medium {
       Nonequilibrium = [Bool]
       Solvent = [String]
       SolverType = [String]
       MatrixSymm = [Bool]
       Correction = [Double]
       DiagonalIntegrator = [String]
       DiagonalScaling = [Double]
       ProbeRadius = [Double]
       Green<GreenTag> {
             Type = [String]
```

(continues on next page)

(continued from previous page)

```
             Der = [String]
             Eps = [Double]
             EpsDyn = [Double]
             Eps1 = [Double]
             EpsDyn1 = [Double]
             Eps2 = [Double]
             EpsDyn2 = [Double]
             Center = [Double]
             Width = [Double]
             InterfaceOrigin = [Array of Doubles]
             MaxL = [Integer]
       }
}
Molecule {
   MEP = [Bool]
   Geometry = [Double]
}
ChargeDistribution {
  Monopoles = [Double]
  Dipoles = [Double]
}
MMFQ {
  SitesPerFragment = [Integer]
  Sites = [Array of Doubles]
  NonPolarizable = [Bool]
}
```

Array-valued keywords will expect the array to be given in comma-separated format and enclosed in square brackets. The purpose of tags is to distinguish between cases in which multiple instances of the same kind of object can be managed by the program. There exist only certain legal tagnames and these are determined in the C++ code. Be aware that the input parsing script does not check the correctness of tags.

1.3 Input parameters

Available sections:

- top section: sets up parameters affecting the module globally;
- Cavity: sets up all information needed to form the cavity and discretize its surface;
- Medium: sets up the solver to be used and the properties of the medium, i.e. the Green's functions inside and outside the cavity;
- Green, subsection of medium. Sets up the Green's function inside and outside the cavity.
- Molecule: molecular geometry to be used in a standalone run.
- ChargeDistribution: sets up a classical multipolar (currently up to dipoles) charge distribution to use as additional source of electrostatic potential.

Note: The Molecule and ChargeDistribution sections only make sense in a standalone run, i.e. when using the `run_pcm` executable.

Warning: Exactly matching results obtained from implementations of IEFPCM and/or CPCM (COSMO) given in other program packages requires careful selection of all the parameters involved. A partial checklist of parameters you should always keep in mind:

- solvent permittivities (static and optical)
- atomic radii set
- scaling of the atomic radii
- cavity surface
- cavity partition (tesselation)
- PCM matrix formation algorithm
- strategy used to solve the PCM linear equations system.

1.3.1 Top section keywords

Units Units of measure used in the input file. If Angstrom is given, all relevant input parameters are first converted in au and subsequently parsed.

- **Type**: string
- **Valid values**: AU | Angstrom
- **Default**: No Default

CODATA Set of fundamental physical constants to be used in the module.

- **Type**: integer
- **Valid values**: 2010 | 2006 | 2002 | 1998
- **Default**: 2010

1.3.2 Cavity section keywords

Type The type of the cavity. Completely specifies type of molecular surface and its discretization. Only one type is allowed. Restart cavity will read the file specified by NpzFile keyword and create a GePol cavity from that.

- **Type**: string
- **Valid values**: GePol | Restart
- **Default**: none

NpzFile The name of the `.npz` file to be used for the GePol cavity restart.

- **Type**: string
- **Default**: empty string

Area Average area (weight) of the surface partition for the GePol cavity.

- **Type**: double
- **Valid values**: $d \geq 0.01\,\mathrm{a.u.}^2$
- **Valid for**: GePol cavity
- **Default value**: $0.3\,\mathrm{a.u.}^2$

Scaling If true, the radii for the spheres will be scaled by 1.2. For finer control on the scaling factor for each sphere, select explicit creation mode.

- **Type**: bool
- **Valid for**: all cavities except Restart
- **Default value**: True

RadiiSet Select set of atomic radii to be used. Currently Bondi-Mantina [Bondi64][MantinaChamberlinValero+09], UFF [RCC+92] and Allinger's MM3 [AZB94] sets available, see *Available radii*.

- **Type**: string
- **Valid values**: Bondi | UFF | Allinger
- **Valid for**: all cavities except Restart
- **Default value**: Bondi

Note: Radii in Allinger's MM3 set are obtained by **dividing** the value in the original paper by 1.2, as done in the ADF COSMO implementation We advise to turn off scaling of the radii by 1.2 when using this set.

MinRadius Minimal radius for additional spheres not centered on atoms. An arbitrarily big value is equivalent to switching off the use of added spheres, which is the default.

- **Type**: double
- **Valid values**: $d \geq 0.4$ a.u.
- **Valid for**: GePol cavity
- **Default value**: 100.0 a.u.

Mode How to create the list of spheres for the generation of the molecular surface:

- in Implicit mode, the atomic coordinates and charges will be obtained from the QM host program. Spheres will be centered on the atoms and the atomic radii, as specified in one the built-in sets, will be used. Scaling by 1.2 will be applied according to the keyword Scaling;
- in Atoms mode, the atomic coordinates and charges will be obtained from the QM host program. For the atoms specified by the array given in keyword Atoms, the built-in radii will be substituted by the radii provided in the keyword Radii. Scaling by 1.2 will be applied according to the keyword Scaling;
- in Explicit mode, both centers and radii of the spheres are to be specified in the keyword Spheres. The user has full control over the generation of the list of spheres. Scaling by 1.2 is **not** applied, regardless of the value of the Scaling keyword.
- **Type**: string
- **Valid values**: Implicit | Atoms | Explicit
- **Valid for**: all cavities except Restart
- **Default value**: Implicit

Atoms Array of atoms whose radius has to be substituted by a custom value.

- **Type**: array of integers
- **Valid for**: all cavities except Restart

Radii Array of radii replacing the built-in values for the selected atoms.

- **Type**: array of doubles

- **Valid for**: all cavities except Restart

Spheres Array of coordinates and centers for construction of the list of spheres in explicit mode. Format is $[\dots, x_i, y_i, z_i, R_i, \dots]$

- **Type**: array of doubles
- **Valid for**: all cavities except Restart

1.3.3 Medium section keywords

SolverType Type of solver to be used. All solvers are based on the Integral Equation Formulation of the Polarizable Continuum Model [CancesMennucci98]

- IEFPCM. Collocation solver for a general dielectric medium
- CPCM. Collocation solver for a conductor-like approximation to the dielectric medium
- **Type**: string
- **Valid values**: IEFPCM | CPCM
- **Default value**: IEFPCM

Nonequilibrium Initializes an additional solver using the dynamic permittivity. To be used in response calculations.

- **Type**: bool
- **Valid for**: all solvers
- **Default value**: False

Solvent Specification of the dielectric medium outside the cavity. This keyword **must always** be given a value. If the solvent name given is different from Explicit any other settings in the Green's function section will be overridden by the built-in values for the solvent specified. See Table *Available solvents* for details. `Solvent = Explicit`, triggers parsing of the Green's function sections.

- **Type**: string
- **Valid values**:
  - Water , H2O;
  - Propylene Carbonate , C4H6O3;
  - Dimethylsulfoxide , DMSO;
  - Nitromethane , CH3NO2;
  - Acetonitrile , CH3CN;
  - Methanol , CH3OH;
  - Ethanol , CH3CH2OH;
  - Acetone , C2H6CO;
  - 1,2-Dichloroethane , C2H4CL2;
  - Methylenechloride , CH2CL2;
  - Tetrahydrofurane , THF;
  - Aniline , C6H5NH2;
  - Chlorobenzene , C6H5CL;
  - Chloroform , CHCL3;

  - Toluene , C6H5CH3;
  - 1,4-Dioxane , C4H8O2;
  - Benzene , C6H6;
  - Carbon Tetrachloride , CCL4;
  - Cyclohexane , C6H12;
  - N-heptane , C7H16;
  - Explicit.

MatrixSymm If True, the PCM matrix obtained by the IEFPCM collocation solver is symmetrized $\mathbf{K} := \frac{\mathbf{K}+\mathbf{K}^\dagger}{2}$

- **Type**: bool
- **Valid for**: IEFPCM solver
- **Default**: True

Correction Correction, k for the apparent surface charge scaling factor in the CPCM solver $f(\varepsilon) = \frac{\varepsilon-1}{\varepsilon+k}$

- **Type**: double
- **Valid values**: $k > 0.0$
- **Valid for**: CPCM solver
- **Default**: 0.0

DiagonalIntegrator Type of integrator for the diagonal of the boundary integral operators

- **Type**: string
- **Valid values**: COLLOCATION
- **Valid for**: IEFPCM, CPCM
- **Default**: COLLOCATION
- **Notes**: in future releases we will add PURISIMA and NUMERICAL as options

DiagonalScaling Scaling factor for diagonal of collocation matrices

- **Type**: double
- **Valid values**: $f > 0.0$
- **Valid for**: IEFPCM, CPCM
- **Default**: 1.07
- **Notes**: values commonly used in the literature are 1.07 and 1.0694

ProbeRadius Radius of the spherical probe approximating a solvent molecule. Used for generating the solvent-excluded surface (SES) or an approximation of it. Overridden by the built-in value for the chosen solvent.

- **Type**: double
- **Valid values**: $d \in [0.1, 100.0]$ a.u.
- **Valid for**: all solvers
- **Default**: 1.0

1.3.4 Green section keywords

If `Solvent = Explicit`, **two** Green's functions sections must be specified with tags `inside` and `outside`, i.e. `Green<inside>` and `Green<outside>`. The Green's function inside will always be the vacuum, while the Green's function outside might vary.

Type Which Green's function characterizes the medium.

- **Type**: string
- **Valid values**: Vacuum | UniformDielectric | SphericalDiffuse | SphericalSharp
- **Default**: Vacuum

Der How to calculate the directional derivatives of the Green's function:

- Numerical, perform numerical differentiation **debug option**;
- Derivative, use automatic differentiation to get the directional derivative;
- Gradient, use automatic differentiation to get the full gradient **debug option**;
- Hessian, use automatic differentiation to get the full hessian **debug option**;
- **Type**: string
- **Valid values**: Numerical | Derivative | Gradient | Hessian
- **Default**: Derivative

Note: The spherical diffuse Green's function **always** uses numerical differentiation.

Eps Static dielectric permittivity of the medium

- **Type**: double
- **Valid values**: $\varepsilon \geq 1.0$
- **Default**: 1.0

EpsDyn Dynamic dielectric permittivity of the medium

- **Type**: double
- **Valid values**: $\varepsilon \geq 1.0$
- **Default**: 1.0

Profile Functional form of the dielectric profile

- **Type**: string
- **Valid values**: Tanh | Erf | Log
- **Valid for**: SphericalDiffuse
- **Default**: Log

Eps1 Static dielectric permittivity inside the interface

- **Type**: double
- **Valid values**: $\varepsilon \geq 1.0$
- **Valid for**: SphericalDiffuse, SphericalSharp
- **Default**: 1.0

EpsDyn1 Dynamic dielectric permittivity inside the interface

- **Type**: double
- **Valid values**: $\varepsilon \geq 1.0$
- **Valid for**: SphericalDiffuse, SphericalSharp
- **Default**: 1.0

Eps2 Static dielectric permittivity outside the interface

- **Type**: double
- **Valid values**: $\varepsilon \geq 1.0$
- **Valid for**: SphericalDiffuse, SphericalSharp
- **Default**: 1.0

EpsDyn2 Dynamic dielectric permittivity outside the interface

- **Type**: double
- **Valid values**: $\varepsilon \geq 1.0$
- **Valid for**: SphericalDiffuse, SphericalSharp
- **Default**: 1.0

Center Center of the interface layer. This corresponds to the radius of the spherical droplet.

- **Type**: double
- **Valid for**: SphericalDiffuse, SphericalSharp
- **Default**: 100.0 a.u.

Width Physical width of the interface layer. This value is divided by 6.0 internally.

- **Type**: double
- **Valid for**: SphericalDiffuse
- **Default**: 5.0 a.u.

> **Warning:** Numerical instabilities may arise if a too small value is selected.

InterfaceOrigin Center of the spherical droplet

- **Type**: array of doubles
- **Valid for**: SphericalDiffuse, SphericalSharp
- **Default**: $[0.0, 0.0, 0.0]$

MaxL Maximum value of the angular momentum in the expansion of the Green's function for the spherical diffuse Green's function

- **Type**: integer
- **Valid for**: SphericalDiffuse, SphericalSharp
- **Default**: 30

1.3.5 Molecule section keywords

It is possible to run the module standalone and use a classical charge distribution as specified in this section of the input. The `run_pcm` executable has to be compiled for a standalone run with:

```
python go_pcm.py -x molecule.inp
```

where the `molecule.inp` input file looks like:

```
units = angstrom
codata = 2002
medium
{
        solvertype = cpcm
        correction = 0.5
    solvent = cyclohexane
}

cavity
{
        type = gepol
        area = 0.6
        radiiset = uff
        mode = implicit
}

molecule
{
    # x, y, z, q
    geometry = [0.000000000, 0.00000000,  0.08729478, 9.0,
                0.000000000, 0.00000000, -1.64558444, 1.0]
}
```

Geometry Coordinates and charges of the molecular aggregate. Format is $[\ldots, x_i, y_i, z_i, Q_i, \ldots]$ Charges are always assumed to be in atomic units

- **Type**: array of doubles

1.3.6 ChargeDistribution section keywords

Set a classical charge distribution, inside or outside the cavity No additional spheres will be generated.

Monopoles Array of point charges Format is $[\ldots, x_i, y_i, z_i, Q_i, \ldots]$

- **Type**: array of doubles

Dipoles Array of point dipoles. Format is $[\ldots, x_i, y_i, z_i, \mu_{x_i}, \mu_{y_i}, \mu_{z_i} \ldots]$ The dipole moment components are always read in atomic units.

- **Type**: array of doubles

1.3.7 MMFQ section keywords

Set a classical fluctuating charge force field. This is incompatible with any options specifying a continuum model. No additional spheres will be generated.

SitesPerFragment Number of sites per MM fragment. For water this is 3.

- **Type**: integer
- **Default**: 3

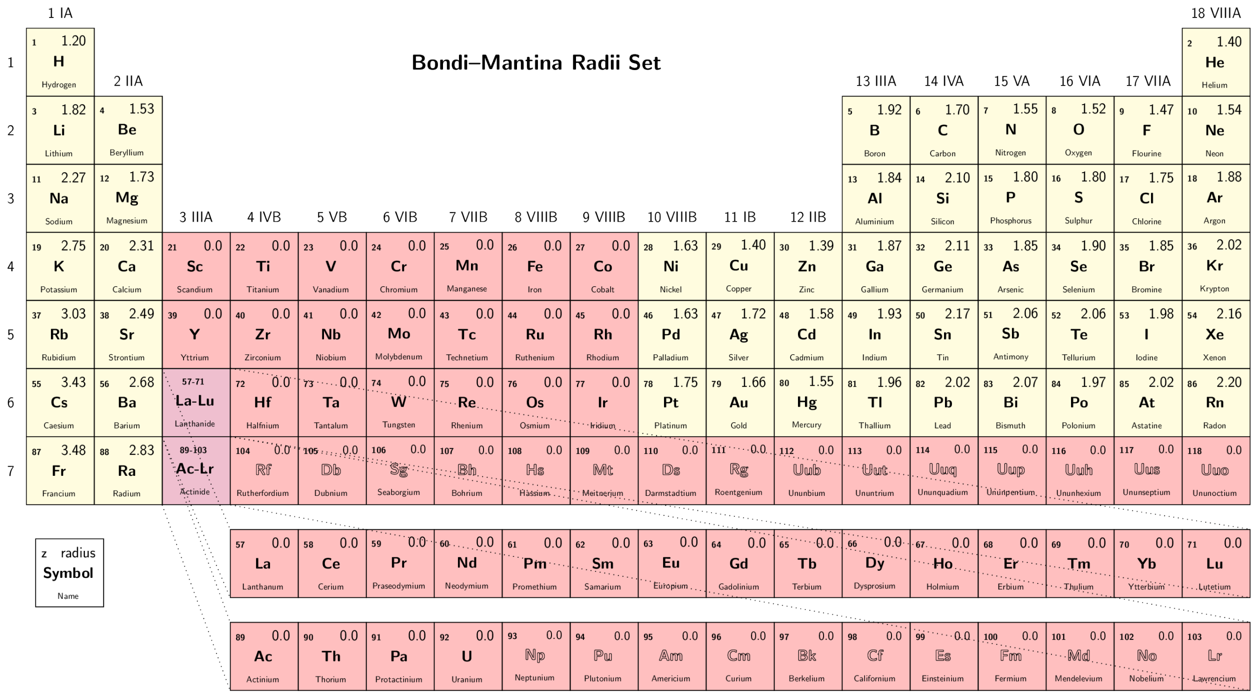
Sites Array of MM sites for the FQ model Format is $[\ldots, x_i, y_i, z_i, chi_i, eta_i \ldots]$

- **Type**: array of doubles

NonPolarizable Whether to make this force field nonpolarizable.

- **Type**: bool
- **Default**: false

1.3.8 Available radii



Bondi–Mantina Radii Set

	1 IA	2 IIA	3 IIIA	4 IVB	5 VB	6 VIB	7 VIIB	8 VIIIB	9 VIIIB	10 VIIIB	11 IB	12 IIB	13 IIIA	14 IVA	15 VA	16 VIA	17 VIIA	18 VIIIA
1	1 1.20 H Hydrogen																	2 1.40 He Helium
2	3 1.82 Li Lithium	4 1.53 Be Beryllium											5 1.92 B Boron	6 1.70 C Carbon	7 1.55 N Nitrogen	8 1.52 O Oxygen	9 1.47 F Flourine	10 1.54 Ne Neon
3	11 2.27 Na Sodium	12 1.73 Mg Magnesium											13 1.84 Al Aluminium	14 2.10 Si Silicon	15 1.80 P Phosphorus	16 1.80 S Sulphur	17 1.75 Cl Chlorine	18 1.88 Ar Argon
4	19 2.75 K Potassium	20 2.31 Ca Calcium	21 0.0 Sc Scandium	22 0.0 Ti Titanium	23 0.0 V Vanadium	24 0.0 Cr Chromium	25 0.0 Mn Manganese	26 0.0 Fe Iron	27 0.0 Co Cobalt	28 1.63 Ni Nickel	29 1.40 Cu Copper	30 1.39 Zn Zinc	31 1.87 Ga Gallium	32 2.11 Ge Germanium	33 1.85 As Arsenic	34 1.90 Se Selenium	35 1.85 Br Bromine	36 2.02 Kr Krypton
5	37 3.03 Rb Rubidium	38 2.49 Sr Strontium	39 0.0 Y Yttrium	40 0.0 Zr Zirconium	41 0.0 Nb Niobium	42 0.0 Mo Molybdenum	43 0.0 Tc Technetium	44 0.0 Ru Ruthenium	45 0.0 Rh Rhodium	46 1.63 Pd Palladium	47 1.72 Ag Silver	48 1.58 Cd Cadmium	49 1.93 In Indium	50 2.17 Sn Tin	51 2.06 Sb Antimony	52 2.06 Te Tellurium	53 1.98 I Iodine	54 2.16 Xe Xenon
6	55 3.43 Cs Caesium	56 2.68 Ba Barium	57-71 La-Lu Lanthanide	72 0.0 Hf Halfnium	73 0.0 Ta Tantalum	74 0.0 W Tungsten	75 0.0 Re Rhenium	76 0.0 Os Osmium	77 0.0 Ir Iridium	78 1.75 Pt Platinum	79 1.66 Au Gold	80 1.55 Hg Mercury	81 1.96 Tl Thallium	82 2.02 Pb Lead	83 2.07 Bi Bismuth	84 1.97 Po Polonium	85 2.02 At Astatine	86 2.20 Rn Radon
7	87 3.48 Fr Francium	88 2.83 Ra Radium	89-103 Ac-Lr Actinide	104 0.0 Rf Rutherfordium	105 0.0 Db Dubnium	106 0.0 Sg Seaborgium	107 0.0 Bh Bohrium	108 0.0 Hs Hassium	109 0.0 Mt Meitnerium	110 0.0 Ds Darmstadtium	111 0.0 Rg Roentgenium	112 0.0 Uub Ununbium	113 0.0 Uut Ununtrium	114 0.0 Uuq Ununquadium	115 0.0 Uup Ununpentium	116 0.0 Uuh Ununhexium	117 0.0 Uus Ununseptium	118 0.0 Uuo Ununoctium
	z radius Symbol Name			57 0.0 La Lanthanum	58 0.0 Ce Cerium	59 0.0 Pr Praseodymium	60 0.0 Nd Neodymium	61 0.0 Pm Promethium	62 0.0 Sm Samarium	63 0.0 Eu Europium	64 0.0 Gd Gadolinium	65 0.0 Tb Terbium	66 0.0 Dy Dysprosium	67 0.0 Ho Holmium	68 0.0 Er Erbium	69 0.0 Tm Thulium	70 0.0 Yb Ytterbium	71 0.0 Lu Lutetium
				89 0.0 Ac Actinium	90 0.0 Th Thorium	91 0.0 Pa Protactinium	92 0.0 U Uranium	93 0.0 Np Neptunium	94 0.0 Pu Plutonium	95 0.0 Am Americium	96 0.0 Cm Curium	97 0.0 Bk Berkelium	98 0.0 Cf Californium	99 0.0 Es Einsteinium	100 0.0 Fm Fermium	101 0.0 Md Mendelevium	102 0.0 No Nobelium	103 0.0 Lr Lawrencium

UFF Radii Set

	1 IA	2 IIA	3 IIIA	4 IVB	5 VB	6 VIB	7 VIIB	8 VIIIB	9 VIIIB	10 VIIIB	11 IB	12 IIB	13 IIIA	14 IVA	15 VA	16 VIA	17 VIIA	18 VIIIA
1	1 1.4430 H Hydrogen																	2 1.81 He Helium
2	3 1.2255 Li Lithium	4 1.3725 Be Beryllium											5 2.0415 B Boron	6 1.9255 C Carbon	7 1.83 N Nitrogen	8 1.75 O Oxygen	9 1.682 F Fluorine	10 1.6215 Ne Neon
3	11 1.4915 Na Sodium	12 1.5105 Mg Magnesium											13 2.2495 Al Aluminium	14 2.1475 Si Silicon	15 2.0735 P Phosphorus	16 2.0175 S Sulphur	17 1.9735 Cl Chlorine	18 1.934 Ar Argon
4	19 1.9060 K Potassium	20 1.6995 Ca Calcium	21 1.6475 Sc Scandium	22 1.5875 Ti Titanium	23 1.5720 V Vanadium	24 1.5115 Cr Chromium	25 1.4805 Mn Manganese	26 1.4560 Fe Iron	27 1.4360 Co Cobalt	28 1.4170 Ni Nickel	29 1.7475 Cu Copper	30 1.3815 Zn Zinc	31 2.1915 Ga Gallium	32 2.14 Ge Germanium	33 2.115 As Arsenic	34 2.1025 Se Selenium	35 2.0945 Br Bromine	36 2.0705 Kr Krypton
5	37 2.0570 Rb Rubidium	38 1.8205 Sr Strontium	39 1.6725 Y Yttrium	40 1.5620 Zr Zirconium	41 1.5825 Nb Niobium	42 1.526 Mo Molybdenum	43 1.499 Tc Technetium	44 1.4815 Ru Ruthenium	45 1.4645 Rh Rhodium	46 1.4495 Pd Palladium	47 1.5740 Ag Silver	48 1.4240 Cd Cadmium	49 2.2315 In Indium	50 2.1960 Sn Tin	51 2.2100 Sb Antimony	52 2.2350 Te Tellurium	53 2.25 I Iodine	54 2.2020 Xe Xenon
6	55 2.2585 Cs Caesium	56 1.8515 Ba Barium	57-71 La-Lu Lanthanide	72 1.5705 Hf Hafnium	73 1.850 Ta Tantalum	74 1.5345 W Tungsten	75 1.4770 Re Rhenium	76 1.5600 Os Osmium	77 1.4200 Ir Iridium	78 1.3770 Pt Platinum	79 1.6465 Au Gold	80 1.3525 Hg Mercury	81 2.1735 Tl Thallium	82 2.1485 Pb Lead	83 2.1850 Bi Bismuth	84 2.3545 Po Polonium	85 2.3750 At Astatine	86 2.3825 Rn Radon
7	87 2.4500 Fr Francium	88 1.8385 Ra Radium	89-103 Ac-Lr Actinide	104 0.0 Rf Rutherfordium	105 0.0 Db Dubnium	106 0.0 Sg Seaborgium	107 0.0 Bh Bohrium	108 0.0 Hs Hassium	109 0.0 Mt Meitnerium	110 0.0 Ds Darmstadtium	111 0.0 Rg Roentgenium	112 0.0 Uub Ununbium	113 0.0 Uut Ununtrium	114 0.0 Uuq Ununquadium	115 0.0 Uup Ununpentium	116 0.0 Uuh Ununhexium	117 0.0 Uus Ununseptium	118 0.0 Uuo Ununoctium
				57 1.7610 La Lanthanum	58 1.7780 Ce Cerium	59 1.8030 Pr Praseodymium	60 1.7875 Nd Neodymium	61 1.7735 Pm Promethium	62 1.7600 Sm Samarium	63 1.7465 Eu Europium	64 1.6840 Gd Gadolinium	65 1.7255 Tb Terbium	66 1.7140 Dy Dysprosium	67 1.7045 Ho Holmium	68 1.6955 Er Erbium	69 1.6870 Tm Thulium	70 1.6775 Yb Ytterbium	71 1.8200 Lu Lutetium
				89 1.7390 Ac Actinium	90 1.6980 Th Thorium	91 1.7120 Pa Protactinium	92 1.6975 U Uranium	93 1.7120 Np Neptunium	94 1.7120 Pu Plutonium	95 1.6905 Am Americium	96 1.6630 Cm Curium	97 1.6695 Bk Berkelium	98 1.6565 Cf Californium	99 1.6495 Es Einsteinium	100 1.6430 Fm Fermium	101 1.6370 Md Mendelevium	102 1.6240 No Nobelium	103 1.6180 Lr Lawrencium

z radius
Symbol
Name

Allinger's MM3 Radii Set

	1 IA	2 IIA	3 IIIA	4 IVB	5 VB	6 VIB	7 VIIB	8 VIIIB	9 VIIIB	10 VIIIB	11 IB	12 IIB	13 IIIA	14 IVA	15 VA	16 VIA	17 VIIA	18 VIIIA
1	1 1.35 H Hydrogen																	2 1.275 He Helium
2	3 2.125 Li Lithium	4 1.85833 Be Beryllium											5 1.79167 B Boron	6 1.70 C Carbon	7 1.60833 N Nitrogen	8 1.51667 O Oxygen	9 1.425 F Fluorine	10 1.33333 Ne Neon
3	11 2.25 Na Sodium	12 2.025 Mg Magnesium											13 1.96667 Al Aluminium	14 1.90833 Si Silicon	15 1.85 P Phosphorus	16 1.79167 S Sulphur	17 1.725 Cl Chlorine	18 1.65833 Ar Argon
4	19 2.575 K Potassium	20 2.34167 Ca Calcium	21 2.175 Sc Scandium	22 1.99167 Ti Titanium	23 1.90833 V Vanadium	24 1.875 Cr Chromium	25 1.86667 Mn Manganese	26 1.85833 Fe Iron	27 1.85833 Co Cobalt	28 1.85 Ni Nickel	29 1.88333 Cu Copper	30 1.90833 Zn Zinc	31 2.05 Ga Gallium	32 2.03333 Ge Germanium	33 1.96667 As Arsenic	34 1.90833 Se Selenium	35 1.85 Br Bromine	36 1.79167 Kr Krypton
5	37 2.70833 Rb Rubidium	38 2.5 Sr Strontium	39 2.25833 Y Yttrium	40 2.11667 Zr Zirconium	41 2.025 Nb Niobium	42 1.99167 Mo Molybdenum	43 1.96667 Tc Technetium	44 1.95 Ru Ruthenium	45 1.95 Rh Rhodium	46 1.975 Pd Palladium	47 2.025 Ag Silver	48 2.08333 Cd Cadmium	49 2.2 In Indium	50 2.15833 Sn Tin	51 2.1 Sb Antimony	52 2.03333 Te Tellurium	53 1.96667 I Iodine	54 1.9 Xe Xenon
6	55 2.86667 Cs Caesium	56 2.55833 Ba Barium	57-71 La-Lu Lanthanide	72 2.10833 Hf Hafnium	73 2.025 Ta Tantalum	74 1.99167 W Tungsten	75 1.975 Re Rhenium	76 1.95833 Os Osmium	77 1.96667 Ir Iridium	78 1.99167 Pt Platinum	79 2.025 Au Gold	80 2.10833 Hg Mercury	81 2.15833 Tl Thallium	82 2.28333 Pb Lead	83 2.21667 Bi Bismuth	84 2.15833 Po Polonium	85 2.09167 At Astatine	86 2.025 Rn Radon
7	87 3.03333 Fr Francium	88 2.725 Ra Radium	89-103 Ac-Lr Actinide	104 2.275 Rf Rutherfordium	105 2.19167 Db Dubnium	106 0.0 Sg Seaborgium	107 1.35 Bh Bohrium	108 0.0 Hs Hassium	109 0.0 Mt Meitnerium	110 0.0 Ds Darmstadtium	111 0.0 Rg Roentgenium	112 0.0 Uub Ununbium	113 0.0 Uut Ununtrium	114 0.0 Uuq Ununquadium	115 0.0 Uup Ununpentium	116 0.0 Uuh Ununhexium	117 0.0 Uus Ununseptium	118 0.0 Uuo Ununoctium
				57 2.31667 La Lanthanum	58 2.28333 Ce Cerium	59 2.275 Pr Praseodymium	60 2.275 Nd Neodymium	61 2.26667 Pm Promethium	62 2.25833 Sm Samarium	63 2.45 Eu Europium	64 2.25833 Gd Gadolinium	65 2.25 Tb Terbium	66 2.24167 Dy Dysprosium	67 2.225 Ho Holmium	68 2.225 Er Erbium	69 2.225 Tm Thulium	70 2.325 Yb Ytterbium	71 2.20833 Lu Lutetium
				89 2.56667 Ac Actinium	90 2.28333 Th Thorium	91 2.2 Pa Protactinium	92 2.1 U Uranium	93 2.1 Np Neptunium	94 2.1 Pu Plutonium	95 0.0 Am Americium	96 0.0 Cm Curium	97 0.0 Bk Berkelium	98 0.0 Cf Californium	99 0.0 Es Einsteinium	100 0.0 Fm Fermium	101 0.0 Md Mendelevium	102 0.0 No Nobelium	103 0.0 Lr Lawrencium

z radius
Symbol
Name

1.3.9 Available solvents

The macroscopic properties for the built-in list of solvents are:

- static permittivity, ε_s
- optical permittivity, ε_∞
- probe radius, r_{probe} in Angstrom.

The following table summarizes the built-in solvents and their properties. Solvents are ordered by decreasing static permittivity.

Name	Formula	ε_s	ε_∞	r_{probe}
Water	H2O	78.39	1.776	1.385
Propylene Carbonate	C4H6O3	64.96	2.019	1.385
Dimethylsulfoxide	DMSO	46.7	2.179	2.455
Nitromethane	CH3NO2	38.20	1.904	2.155
Acetonitrile	CH3CN	36.64	1.806	2.155
Methanol	CH3OH	32.63	1.758	1.855
Ethanol	CH3CH2OH	24.55	1.847	2.180
Acetone	C2H6CO	20.7	1.841	2.38
1,2-Dichloroethane	C2H4Cl2	10.36	2.085	2.505
Methylenechloride	CH2Cl2	8.93	2.020	2.27
Tetrahydrofurane	THF	7.58	1.971	2.9
Aniline	C6H5NH2	6.89	2.506	2.80
Chlorobenzene	C6H5Cl	5.621	2.320	2.805
Chloroform	CHCl3	4.90	2.085	2.48
Toluene	C6H5CH3	2.379	2.232	2.82
1,4-Dioxane	C4H8O2	2.250	2.023	2.630
Benzene	C6H6	2.247	2.244	2.630
Carbon tetrachloride	CCl4	2.228	2.129	2.685
Cyclohexane	C6H12	2.023	2.028	2.815
N-heptane	C7H16	1.92	1.918	3.125

1.4 Interfacing a QM program and PCMSolver

1.4.1 For the impatients: tl;dr

In these examples, we want to show how *every function* in the API works. If your program is written in Fortran, head over to *Interfacing with a Fortran host* If your program is written in C/C++, head over to *Interfacing with a C host*

1.4.2 How PCMSolver handles potentials and charges: surface functions

Electrostatic potential vectors and the corresponding apparent surface charge vectors are handled internally as *surface functions*. The actual values are stored into Eigen vectors and saved into a map. The mapping is between the name of the surface function, given by the programmer writing the interface to the library, and the vector holding the values.

1.4.3 What you should care about: API functions

These are the contents of the `pcmsolver.h` file defining the public API of the PCMSolver library. The Fortran bindings for the API are in the `pcmsolver.f90` file. The indexing of symmetry operations and their mapping to a bitstring is explained in the following Table. This is important when passing symmetry information to the `pcmsolver_new()` function.

Table 1: Symmetry operations indexing within the module

Index	zyx	Generator	Parity
0	000	E	1.0
1	001	Oyz	-1.0
2	010	Oxz	-1.0
3	011	C2z	1.0
4	100	Oxy	-1.0
5	101	C2y	1.0
6	110	C2x	1.0
7	111	i	-1.0

C API to PCMSolver.

Author Roberto Di Remigio

Date 2015

Defines

PCMSolver_EXPORT

pcmsolver_bool_t_DEFINED

Typedefs

typedef bool **pcmsolver_bool_t**

typedef struct pcmsolver_context_s **pcmsolver_context_t**

Workaround to have pcmsolver_context_t available to C

typedef void (***HostWriter**)(**const** char *message)

Flushes module output to host program

Parameters

- `[inout] message`: contents of the module output

Enums

enum pcmsolver_reader_t

Input processing strategies.

Values:

enumerator PCMSOLVER_READER_OWN

Module reads input on its own

enumerator PCMSOLVER_READER_HOST

Module receives input from host

Functions

pcmsolver_context_t ***pcmsolver_new**(*pcmsolver_reader_t* *input_reading*, int *nr_nuclei*, double *charges*[], double *coordinates*[], int *symmetry_info*[], **struct** *PCMInput* **host_input*, *HostWriter* *writer*)

Creates a new PCM context object.

The molecular point group information is passed as an array of 4 integers: number of generators, first, second and third generator respectively. Generators map to integers as in table :ref: `symmetry-ops`

Parameters

- `[in] input_reading`: input processing strategy
- `[in] nr_nuclei`: number of atoms in the molecule
- `[in] charges`: atomic charges
- `[in] coordinates`: atomic coordinates
- `[in] symmetry_info`: molecular point group information
- `[in] host_input`: input to the module, as read by the host
- `[in] writer`: flush-to-host function

pcmsolver_context_t ***pcmsolver_new_v1112**(*pcmsolver_reader_t* *input_reading*, int *nr_nuclei*, double *charges*[], double *coordinates*[], int *symmetry_info*[], **const** char **parsed_fname*, **struct** *PCMInput* **host_input*, *HostWriter* *writer*)

Creates a new PCM context object, updated in v1.1.12.

The molecular point group information is passed as an array of 4 integers: number of generators, first, second and third generator respectively. Generators map to integers as in table :ref: `symmetry-ops`

Parameters

- `[in] input_reading`: input processing strategy
- `[in] nr_nuclei`: number of atoms in the molecule
- `[in] charges`: atomic charges
- `[in] coordinates`: atomic coordinates
- `[in] symmetry_info`: molecular point group information
- `[in] parsed_fname`: name of the input file parsed by pcmsolver.py
- `[in] host_input`: input to the module, as read by the host
- `[in] writer`: flush-to-host function

pcmsolver_context_t ***pcmsolver_new_read_host**(int *nr_nuclei*, double *charges*[], double *coordinates*[], int *symmetry_info*[], *HostWriter* *writer*)

Creates a new PCM context object, with deferred input parsing from host.

The molecular point group information is passed as an array of 4 integers: number of generators, first, second and third generator respectively. Generators map to integers as in table :ref: `symmetry-ops`

Note Added in v1.3.0

Parameters

- `[in] nr_nuclei`: number of atoms in the molecule
- `[in] charges`: atomic charges

- `[in] coordinates`: atomic coordinates
- `[in] symmetry_info`: molecular point group information
- `[in] writer`: flush-to-host function

void **`pcmsolver_set_bool_option`** (*pcmsolver_context_t *context*, **`const`** char **parameter*, *pcmsolver_bool_t value*)

Set a bool option in PCMSolver input.

Warning You should call pcmsolver_refresh to finalize the context object.

Parameters

- `[inout] context`: the PCM context object
- `[in] parameter`: the name of the parameter to set
- `[in] value`: the value of the parameter

void **`pcmsolver_set_int_option`** (*pcmsolver_context_t *context*, **`const`** char **parameter*, int *value*)

Set an integer option in PCMSolver input.

Warning You should call pcmsolver_refresh to finalize the context object.

Parameters

- `[inout] context`: the PCM context object
- `[in] parameter`: the name of the parameter to set
- `[in] value`: the value of the parameter

void **`pcmsolver_set_double_option`** (*pcmsolver_context_t *context*, **`const`** char **parameter*, double *value*)

Set a double option in PCMSolver input.

Warning You should call pcmsolver_refresh to finalize the context object.

Parameters

- `[inout] context`: the PCM context object
- `[in] parameter`: the name of the parameter to set
- `[in] value`: the value of the parameter

void **`pcmsolver_set_string_option`** (*pcmsolver_context_t *context*, **`const`** char **parameter*, **`const`** char **value*)

Set a string option in PCMSolver input.

Warning You should call pcmsolver_refresh to finalize the context object.

Parameters

- `[inout] context`: the PCM context object
- `[in] parameter`: the name of the parameter to set
- `[in] value`: the value of the parameter

void **`pcmsolver_refresh`** (*pcmsolver_context_t *context*)

Refreshes the PCM context object.

Parameters

- `[inout] context`: the PCM context object

void **`pcmsolver_delete`** (*pcmsolver_context_t *context*)

Deletes a PCM context object.

Parameters

- `[inout] context`: the PCM context object to be deleted

pcmsolver_bool_t **`pcmsolver_is_compatible_library`** (void)

Whether the library is compatible with the header file Checks that the compiled library and header file version match. Host should abort when that is not the case.

Warning This function should be called **before** instantiating any PCM context objects.

void **`pcmsolver_print`** (*pcmsolver_context_t *context*)

Prints set up information.

Parameters

- `[inout] context`: the PCM context object

void **`pcmsolver_citation`** (*HostWriter writer*)

Print version information and citation for PCMSolver.

Parameters

- `[in] writer`: flush-to-host function

int **`pcmsolver_get_cavity_size`** (*pcmsolver_context_t *context*)

Getter for the number of finite elements composing the molecular cavity.

Return the size of the cavity

Parameters

- `[inout] context`: the PCM context object

int **`pcmsolver_get_irreducible_cavity_size`** (*pcmsolver_context_t *context*)

Getter for the number of irreducible finite elements composing the molecular cavity.

Return the number of irreducible finite elements

Parameters

- `[inout] context`: the PCM context object

void **`pcmsolver_get_centers`** (*pcmsolver_context_t *context*, double *centers*[])

Getter for the centers of the finite elements composing the molecular cavity.

Parameters

- `[inout] context`: the PCM context object
- `[out] centers`: array holding the coordinates of the finite elements centers

void **pcmsolver_get_center** (*pcmsolver_context_t* **context*, int *its*, double *center*[])

Getter for the center of the i-th finite element.

Parameters

- `[inout] context`: the PCM context object
- `[in] its`: index of the finite element
- `[out] center`: array holding the coordinates of the finite element center

void **pcmsolver_get_areas** (*pcmsolver_context_t* **context*, double *areas*[])

Getter for the areas/weights of the finite elements.

Parameters

- `[inout] context`: the PCM context object
- `[out] areas`: array holding the weights/areas of the finite elements

void **pcmsolver_compute_asc** (*pcmsolver_context_t* **context*, **const** char **mep_name*, **const** char **asc_name*, int *irrep*)

Computes ASC given a MEP and the desired irreducible representation.

Parameters

- `[inout] context`: the PCM context object
- `[in] mep_name`: label of the MEP surface function
- `[in] asc_name`: label of the ASC surface function
- `[in] irrep`: index of the desired irreducible representation The module uses the surface function concept to handle potentials and charges. Given labels for each, this function retrieves the MEP and computes the corresponding ASC.

void **pcmsolver_compute_response_asc** (*pcmsolver_context_t* **context*, **const** char **mep_name*, **const** char **asc_name*, int *irrep*)

Computes response ASC given a MEP and the desired irreducible representation.

Parameters

- `[inout] context`: the PCM context object
- `[in] mep_name`: label of the MEP surface function
- `[in] asc_name`: label of the ASC surface function
- `[in] irrep`: index of the desired irreducible representation If `Nonequilibrium = True` in the input, calculates a response ASC using the dynamic permittivity. Falls back to the solver with static permittivity otherwise.

double **pcmsolver_compute_polarization_energy** (*pcmsolver_context_t* **context*, **const** char **mep_name*, **const** char **asc_name*)

Computes the polarization energy.

Return the polarization energy This function calculates the dot product of the given MEP and ASC vectors.

Parameters

- `[inout] context`: the PCM context object

- `[in] mep_name`: label of the MEP surface function
- `[in] asc_name`: label of the ASC surface function

double **`pcmsolver_get_asc_dipole`**(*pcmsolver_context_t *context*, **`const`** char **asc_name*, double *dipole*[])

Getter for the ASC dipole.

Return the ASC dipole, i.e. $\sqrt{\sum_i \mu_i^2}$

Parameters

- `[inout] context`: the PCM context object
- `[in] asc_name`: label of the ASC surface function
- `[out] dipole`: the Cartesian components of the ASC dipole moment

void **`pcmsolver_get_surface_function`**(*pcmsolver_context_t *context*, int *size*, double *values*[], **`const`** char **name*)

Retrieves data wrapped in a given surface function.

Parameters

- `[inout] context`: the PCM context object
- `[in] size`: the size of the surface function
- `[in] values`: the values wrapped in the surface function
- `[in] name`: label of the surface function

void **`pcmsolver_set_surface_function`**(*pcmsolver_context_t *context*, int *size*, double *values*[], **`const`** char **name*)

Sets a surface function given data and label.

Parameters

- `[inout] context`: the PCM context object
- `[in] size`: the size of the surface function
- `[in] values`: the values to be wrapped in the surface function
- `[in] name`: label of the surface function

void **`pcmsolver_print_surface_function`**(*pcmsolver_context_t *context*, **`const`** char **name*)

Prints surface function contents to host output.

Parameters

- `[inout] context`: the PCM context object
- `[in] name`: label of the surface function

void **`pcmsolver_save_surface_functions`**(*pcmsolver_context_t *context*)

Dumps all currently saved surface functions to NumPy arrays in .npy files.

Parameters

- `[inout] context`: the PCM context object void **`pcmsolver_save_surface_function`**(*pcmsolver_context_t *context*, **`const`** char **name*)

Dumps a surface function to NumPy array in .npy file.

Note The name parameter is the name of the NumPy array file **without** .npy extension

Parameters

- `[inout] context`: the PCM context object
- `[in] name`: label of the surface function

void **`pcmsolver_load_surface_function`**(*pcmsolver_context_t *context*, **`const`** char **name*)

Loads a surface function from a .npy file.

Note The name parameter is the name of the NumPy array file **without** .npy extension

Parameters

- `[inout] context`: the PCM context object
- `[in] name`: label of the surface function

void **`pcmsolver_write_timings`**(*pcmsolver_context_t *context*)

Writes timing results for the API.

Parameters

- `[inout] context`: the PCM context object

1.4.4 Host input forwarding

`struct PCMInput`

Data structure for host-API input communication.

Forward-declare *PCMInput* input wrapping struct

Public Members

char **`cavity_type`**[8]

Type of cavity requested.

int **`patch_level`**

Wavelet cavity mesh patch level.

double **`coarsity`**

Wavelet cavity mesh coarsity.

double **`area`**

Average tesserae area.

char **`radii_set`**[8]

The built-in radii set to be used.

double **`min_distance`**

Minimal distance between sampling points.

int **`der_order`**

Derivative order for the switching function.

pcmsolver_bool_t **scaling**
Whether to scale or not the atomic radii.

char **restart_name**[20]
Name of the .npz file for GePol cavity restart.

double **min_radius**
Minimal radius for the added spheres.

char **solver_type**[7]
Type of solver requested.

double **correction**
Correction in the CPCM apparent surface charge scaling factor.

char **solvent**[16]
Name of the solvent.

double **probe_radius**
Radius of the spherical probe mimicking the solvent.

char **equation_type**[11]
Type of the integral equation to be used.

char **inside_type**[7]
Type of Green's function requested inside the cavity.

double **outside_epsilon**
Value of the static permittivity outside the cavity.

char **outside_type**[22]
Type of Green's function requested outside the cavity.

1.4.5 Internal details of the API

class *pcm*::**Meddle**
Contains functions exposing an interface to the module internals.

Author Roberto Di Remigio

Date 2015-2017

Public Functions

Meddle(**const** *Input* &*input*, **const** *HostWriter* &*writer*)
CTOR from *Input* object.

Warning This CTOR is meant to be used with the standalone executable only

Parameters

- `[in] input`: an *Input* object
- `[in] writer`: the global HostWriter object

Meddle(**const** std::string &*inputFileName*, **const** *HostWriter* &*writer*)
CTOR from own input reader.

Warning This CTOR is meant to be used with the standalone executable only

Parameters

- `[in] inputFileName`: name of the parsed, machine-readable input file
- `[in] writer`: the global HostWriter object

Meddle(int *nr_nuclei*, double *charges*[], double *coordinates*[], int *symmetry_info*[], **const** *HostWriter* &*writer*, **const** std::string &*inputFileName*)

CTOR from parsed input file name.

Parameters

- `[in] inputFileName`: name of the parsed, machine-readable input file
- `[in] nr_nuclei`: number of atoms in the molecule
- `[in] charges`: atomic charges
- `[in] coordinates`: atomic coordinates
- `[in] symmetry_info`: molecular point group information
- `[in] writer`: the global HostWriter object

Meddle(int *nr_nuclei*, double *charges*[], double *coordinates*[], int *symmetry_info*[], **const** *PCMInput* &*host_input*, **const** *HostWriter* &*writer*)

Constructor.

The molecular point group information is passed as an array of 4 integers: number of generators, first, second and third generator respectively. Generators map to integers as in table :ref: `symmetry-ops`

Parameters

- `[in] nr_nuclei`: number of atoms in the molecule
- `[in] charges`: atomic charges
- `[in] coordinates`: atomic coordinates
- `[in] symmetry_info`: molecular point group information
- `[in] host_input`: input to the module, as read by the host
- `[in] writer`: the global HostWriter object

Meddle(int *nr_nuclei*, double *charges*[], double *coordinates*[], int *symmetry_info*[], **const** *HostWriter* &*writer*)

Constructor.

The molecular point group information is passed as an array of 4 integers: number of generators, first, second and third generator respectively. Generators map to integers as in table :ref: `symmetry-ops`

Parameters

- `[in] nr_nuclei`: number of atoms in the molecule
- `[in] charges`: atomic charges
- `[in] coordinates`: atomic coordinates
- `[in] symmetry_info`: molecular point group information
- `[in] writer`: the global HostWriter object

Molecule **molecule**() **const**

Getter for the molecule object.

PCMSolverIndex **getCavitySize**() **const**

Getter for the number of finite elements composing the molecular cavity.

Return the size of the cavity

PCMSolverIndex **getIrreducibleCavitySize**() **const**

Getter for the number of irreducible finite elements composing the molecular cavity.

Return the number of irreducible finite elements

void **getCenters**(double *centers*[]) **const**

Getter for the centers of the finite elements composing the molecular cavity.

Parameters

- `[out] centers`: array holding the coordinates of the finite elements centers

void **getCenter**(int *its*, double *center*[]) **const**

Getter for the center of the i-th finite element.

Parameters

- `[in] its`: index of the finite element
- `[out] center`: array holding the coordinates of the finite element center

Eigen::Matrix3Xd **getCenters**() **const**

Getter for the centers of the finite elements composing the molecular cavity.

Return a matrix with the finite elements centers (dimensions 3 x number of finite elements)

void **getAreas**(double *areas*[]) **const**

Getter for the areas/weights of the finite elements.

Parameters

- `[out] areas`: array holding the weights/areas of the finite elements

void **computeASC**(**const** std::string *&mep_name*, **const** std::string *&asc_name*, int *irrep*)

Computes ASC given a MEP and the desired irreducible representation.

Parameters

- `[in] mep_name`: label of the MEP surface function
- `[in] asc_name`: label of the ASC surface function
- `[in] irrep`: index of the desired irreducible representation The module uses the surface function concept to handle potentials and charges. Given labels for each, this function retrieves the MEP and computes the corresponding ASC.

void **computeResponseASC**(**const** std::string *&mep_name*, **const** std::string *&asc_name*, int *irrep*)

Computes response ASC given a MEP and the desired irreducible representation.

Parameters

- `[in] mep_name`: label of the MEP surface function

- `[in] asc_name`: label of the ASC surface function
- `[in] irrep`: index of the desired irreducible representation If `Nonequilibrium = True` in the input, calculates a response ASC using the dynamic permittivity. Falls back to the solver with static permittivity otherwise.

double **`computePolarizationEnergy`**(**`const`** std::string *&mep_name*, **`const`** std::string *&asc_name*) **`const`**

Computes the polarization energy.

Return the polarization energy This function calculates the dot product of the given MEP and ASC vectors.

Parameters

- `[in] mep_name`: label of the MEP surface function
- `[in] asc_name`: label of the ASC surface function

double **`getASCDipole`**(**`const`** std::string *&asc_name*, double *dipole*[]) **`const`**

Getter for the ASC dipole.

Return the ASC dipole, i.e. $\sqrt{\sum_i \mu_i^2}$

Parameters

- `[in] asc_name`: label of the ASC surface function
- `[out] dipole`: the Cartesian components of the ASC dipole moment

void **`getSurfaceFunction`**(PCMSolverIndex *size*, double *values*[], **`const`** std::string *&name*) **`const`**

Retrieves data wrapped in a given surface function.

Parameters

- `[in] size`: the size of the surface function
- `[in] values`: the values wrapped in the surface function
- `[in] name`: label of the surface function

void **`setSurfaceFunction`**(PCMSolverIndex *size*, double *values*[], **`const`** std::string *&name*)

Sets a surface function given data and label.

Parameters

- `[in] size`: the size of the surface function
- `[in] values`: the values to be wrapped in the surface function
- `[in] name`: label of the surface function

void **`printSurfaceFunction`**(**`const`** std::string *&name*) **`const`**

Prints surface function contents to host output.

Parameters

- `[in] name`: label of the surface function void **saveSurfaceFunctions**() **const**

Dumps all currently saved surface functions to NumPy arrays in .npy files.

void **saveSurfaceFunction**(**const** std::string &*name*) **const**

Dumps a surface function to NumPy array in .npy file.

Note The name parameter is the name of the NumPy array file **without** .npy extension

Parameters

- `[in] name`: label of the surface function

void **loadSurfaceFunction**(**const** std::string &*name*)

Loads a surface function from a .npy file.

Note The name parameter is the name of the NumPy array file **without** .npy extension

Parameters

- `[in] name`: label of the surface function

void **printInfo**() **const**

Prints set up information.

std::string **printCitation**() **const**

Prints citation.

void **writeTimings**() **const**

Writes timing results for the API.

Private Functions

void **CTORBody**()

Common implemenation for the CTOR-s

void **initInput**(int *nr_nuclei*, double *charges*[], double *coordinates*[], int *symmetry_info*[], bool *deferred_init* = false)

Initialize input_.

Parameters

- `[in] nr_nuclei`: number of atoms in the molecule
- `[in] charges`: atomic charges
- `[in] coordinates`: atomic coordinates
- `[in] symmetry_info`: molecular point group information
- `[in] deferred_init`: whether to defer initialization of *Molecule*

void **initCavity**()

Initialize cavity_

void **initStaticSolver**()

Initialize static solver K_0_

void **initDynamicSolver**()

Initialize dynamic solver K_d_

void **initMMFQ**()
: Initialize fluctuating charges solver FQ_

void **mediumInfo**(*IGreensFunction* **gf_i*, *IGreensFunction* **gf_o*)
: Collect info on medium

void **GaussCheck**() **const**
: Perform Gauss' theorem check

Private Members

Printer **hostWriter_**
: Output redirect-or to host program output

Input **input_**
: *Input* object

PCMInput **host_input_**
: Host input struct

ICavity ***cavity_**
: Cavity

std::tuple<PCMSolverIndex, PCMSolverIndex> **size_**
: Number of reducible and irreducible classical sites

ISolver ***K_0_**
: Solver with static permittivity

ISolver ***K_d_**
: Solver with dynamic permittivity

mmfq::FQOhno ***FQ_**
: Fluctuating charges solver with Ohno kernel

bool **hasDynamic_**
: Whether K_d_ was initialized

bool **hasFQ_**
: Whether FQ_ was initialized

std::ostringstream **infoStream_**
: PCMSolver set up information

SurfaceFunctionMap **functions_**
: SurfaceFunction map

struct Printer

class *pcm*::**Input**

A wrapper class for the Getkw Library C++ bindings.

An *Input* object is to be used as the unique point of access to user-provided input: input > parsed input (Python script) > *Input* object (contains all the input data) Definition of input parameters is to be done in the Python script and in this class. They must be specified as private data members with public accessor methods (get-ters). Most of the data members are anyway accessed through the input wrapping struct-s In general, no mutator methods (set-ters) should be needed, exceptions to this rule should be carefully considered.

Author Roberto Di Remigio

Date 2013

Public Functions

Input ()
: Default constructor.

Input (**const** std::string &*filename*)
: Constructor from human-readable input file name.

Input (**const** *PCMInput* &*host_input*)
: Constructor from host input structs.

std::string **units** () **const**
: Accessor methods.

  Top-level section input

bool **scaling** () **const**
: Cavity section input.

void **molecule** (**const** *Molecule* &*m*)
: This method sets the molecule and the list of spheres.

Solvent **solvent** () **const**
: Medium section input.

std::string **providedBy** () **const**
: Keeps track of who did the parsing: the API or the host program.

CavityData **cavityParams** () **const**
: Get-ters for input wrapping structs.

Private Functions

void **reader** (**const** *PCMInput* &*host_input*)
: Read host data structures (host-side syntactic input parsing) into *Input* object. It provides access to a **limited** number of options only, basically the ones that can be filled into the cavityInput, solverInput and greenInput data structures. Lengths and areas are **expected** to be in Angstrom/Angstrom^2 and will hence be converted to au/au^2.

  Note Specification of the solvent by name overrides any input given through the greenInput data structure!

  Warning The cavity can only be built in the "Implicit" mode, i.e. by grabbing the coordinates for the sphere centers from the host program. Atomic coordinates are **expected** to be in au! The "Atoms" and "Explicit" methods are only available using the explicit parsing by our Python script of a separate input file.

void **semanticCheck** ()
: Perform semantic input parsing aka sanity check

Private Members

std::string **units_**
Units of measure.

int **CODATAyear_**
Year of the CODATA set to be used.

std::string **cavityType_**
The type of cavity.

std::string **cavFilename_**
Filename for the .npz cavity restart file.

double **area_**
GePol cavity average element area.

bool **scaling_**
Whether the radii should be scaled by 1.2.

std::string **radiiSet_**
The set of radii to be used.

std::string **radiiSetName_**
Collects info on atomic radii set.

double **minimalRadius_**
Minimal radius of an added sphere.

std::string **mode_**
How the API should get the coordinates of the sphere centers.

std::vector<int> **atoms_**
List of selected atoms with custom radii.

std::vector<double> **radii_**
List of radii attached to the selected atoms.

std::vector<Sphere> **spheres_**
List of spheres for fully custom cavity generation.

Molecule **molecule_**
Molecule or atomic aggregate.

Solvent **solvent_**
The solvent for a vacuum/uniform dielectric run.

bool **hasSolvent_**
Whether the medium was initialized from a solvent object.

std::string **solverType_**
The solver type.

double **correction_**
Correction factor (C-PCM)

bool **hermitivitize_**
Whether the PCM matrix should be hermitivitized (collocation solvers)

bool **isDynamic_**
Whether the dynamic PCM matrix should be used.

double **probeRadius_**
Solvent probe radius.

std::string **integratorType_**
Type of integrator for the diagonal of the boundary integral operators.

double **integratorScaling_**
Scaling factor for the diagonal of the approximate collocation boundary integral operators

std::string **greenInsideType_**
The Green's function type inside the cavity. It encodes the Green's function type, derivative calculation strategy and dielectric profile: TYPE_DERIVATIVE_PROFILE

std::string **greenOutsideType_**
The Green's function type outside the cavity It encodes the Green's function type, derivative calculation strategy and dielectric profile: TYPE_DERIVATIVE_PROFILE

double **epsilonInside_**
Permittivity inside the cavity.

double **epsilonStaticOutside_**
Static permittivity outside the cavity.

double **epsilonDynamicOutside_**
Dynamic permittivity outside the cavity.

double **epsilonStatic1_**
Diffuse interface: static permittivity inside the interface.

double **epsilonDynamic1_**
Diffuse interface: dynamic permittivity inside the interface.

double **epsilonStatic2_**
Diffuse interface: static permittivity outside the interface.

double **epsilonDynamic2_**
Diffuse interface: dynamic permittivity outside the interface.

double **center_**
Center of the diffuse interface.

double **width_**
Width of the diffuse interface.

int **maxL_**
Maximum angular momentum.

std::vector<double> **origin_**
Center of the dielectric sphere.

std::vector<double> **geometry_**
Molecular geometry.

bool **isFQ_**
Whether this is a FQ calculation.

bool **isNonPolarizable_**
Whether this is a nonpolarizable MM calculation.

bool **MEPfromMolecule_**
Whether to calculate the MEP from the molecular geometry.

bool **MEPfromChargeDist_**
Whether to calculate the MEP from the charge distribution.

ChargeDistribution **multipoles_**
Classical charge distribution of point multipoles.

MMFQ **fragments_**
: Classical fluctuating charges MM force field.

std::string **providedBy_**
: Who performed the syntactic input parsing.

Friends

friend std::ostream &**operator<<** (std::ostream &*os*, **const** *Input* &*input*)
: Operators operator<<

1.5 Interfacing with a Fortran host

```fortran
!
! PCMSolver, an API for the Polarizable Continuum Model
! Copyright (C) 2020 Roberto Di Remigio, Luca Frediani and contributors.
!
! This file is part of PCMSolver.
!
! PCMSolver is free software: you can redistribute it and/or modify
! it under the terms of the GNU Lesser General Public License as published by
! the Free Software Foundation, either version 3 of the License, or
! (at your option) any later version.
!
! PCMSolver is distributed in the hope that it will be useful,
! but WITHOUT ANY WARRANTY; without even the implied warranty of
! MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
! GNU Lesser General Public License for more details.
!
! You should have received a copy of the GNU Lesser General Public License
! along with PCMSolver.  If not, see <http://www.gnu.org/licenses/>.
!
! For information on the complete list of contributors to the
! PCMSolver API, see: <http://pcmsolver.readthedocs.io/>
!

program pcm_fortran_host

  use, intrinsic :: iso_c_binding
  use, intrinsic :: iso_fortran_env, only: output_unit, error_unit
  use pcmsolver
  use utilities
  use testing

  implicit none

  type(c_ptr) :: pcm_context
  integer(c_int) :: nr_nuclei
  real(c_double), allocatable :: charges(:)
  real(c_double), allocatable :: coordinates(:)
  integer(c_int) :: symmetry_info(4)
  type(PCMInput) :: host_input
  logical :: log_open, log_exist
  character(kind=c_char, len=*), parameter :: mep_lbl = 'NucMEP'
  character(kind=c_char, len=*), parameter :: asc_lbl = 'NucASC'
```

(continues on next page)

(continued from previous page)

```fortran
  character(kind=c_char, len=*), parameter :: asc_B3g_lbl = 'OITASC'
  character(kind=c_char, len=*), parameter :: asc_neq_B3g_lbl = 'ASCB3g'
  real(c_double), allocatable :: grid(:), mep(:), asc_Ag(:), asc_B3g(:), asc_neq_B3g(:), areas(:)
  integer(c_int) :: grid_size, irr_grid_size
  real(c_double) :: energy
  ! Reference values for scalar quantities
  integer(c_int), parameter :: ref_size = 576, ref_irr_size = 72
  real(c_double), parameter :: ref_energy = -0.437960027982

  if (.not. pcmsolver_is_compatible_library()) then
    write (error_unit, *) 'PCMSolver library not compatible!'
    stop
  end if

  ! Open a file for the output...
  inquire (file='Fortran_host.out', opened=log_open, &
           exist=log_exist)
  if (log_exist) then
    open (unit=output_unit, &
          file='Fortran_host.out', &
          status='unknown', &
          form='formatted', &
          access='sequential')
    close (unit=output_unit, status='delete')
  end if
  open (unit=output_unit, &
        file='Fortran_host.out', &
        status='new', &
        form='formatted', &
        access='sequential')
  rewind (output_unit)
  write (output_unit, *) 'Starting a PCMSolver calculation'
  call pcmsolver_citation(c_funloc(host_writer))

  nr_nuclei = 6_c_int
  allocate (charges(nr_nuclei))
  allocate (coordinates(3*nr_nuclei))

  ! Use C2H4 in D2h symmetry
  charges = (/6.0_c_double, 1.0_c_double, 1.0_c_double, &
              6.0_c_double, 1.0_c_double, 1.0_c_double/)
  coordinates = (/0.0_c_double, 0.0_c_double, 1.257892_c_double, &
                  0.0_c_double, 1.745462_c_double, 2.342716_c_double, &
                  0.0_c_double, -1.745462_c_double, 2.342716_c_double, &
                  0.0_c_double, 0.0_c_double, -1.257892_c_double, &
                  0.0_c_double, 1.745462_c_double, -2.342716_c_double, &
                  0.0_c_double, -1.745462_c_double, -2.342716_c_double/)

  ! This means the molecular point group has three generators:
  ! the Oxy, Oxz and Oyz planes
  symmetry_info = (/3, 4, 2, 1/)

  host_input = pcmsolver_fill_pcminput(area=.2d0, scaling=.true., solver_type='iefpcm', solvent='water')

  pcm_context = pcmsolver_new(PCMSOLVER_READER_HOST, &
```

(continues on next page)

(continued from previous page)

```
                              nr_nuclei, charges, coordinates, &
                              symmetry_info, host_input, &
                              c_funloc(host_writer))

  call pcmsolver_print(pcm_context)

  grid_size = pcmsolver_get_cavity_size(pcm_context)
  irr_grid_size = pcmsolver_get_irreducible_cavity_size(pcm_context)
  allocate (grid(3*grid_size))
  grid = 0.0_c_double
  call pcmsolver_get_centers(pcm_context, grid)
  allocate (areas(grid_size))
  call pcmsolver_get_areas(pcm_context, areas)

  allocate (mep(grid_size))
  mep = 0.0_c_double
  mep = nuclear_mep(nr_nuclei, charges, reshape(coordinates, (/3_c_int, nr_nuclei/)), &
                    grid_size, reshape(grid, (/3_c_int, grid_size/)))
  call pcmsolver_set_surface_function(pcm_context, grid_size, mep, mep_lbl)
  ! This is the Ag irreducible representation (totally symmetric)
  call pcmsolver_compute_asc(pcm_context, &
                             mep_lbl, &
                             asc_lbl, &
                             irrep=0_c_int)
  allocate (asc_Ag(grid_size))
  asc_Ag = 0.0_c_double
  call pcmsolver_get_surface_function(pcm_context, grid_size, asc_Ag, asc_lbl)

  energy = pcmsolver_compute_polarization_energy(pcm_context, &
                                                 mep_lbl, &
                                                 asc_lbl)

  write (output_unit, '(A, F20.12)') 'Polarization energy = ', energy

  allocate (asc_neq_B3g(grid_size))
  asc_neq_B3g = 0.0_c_double
  ! This is the B3g irreducible representation
  call pcmsolver_compute_response_asc(pcm_context, &
                                      mep_lbl, &
                                      asc_neq_B3g_lbl, &
                                      irrep=3_c_int)
  call pcmsolver_get_surface_function(pcm_context, grid_size, asc_neq_B3g, asc_neq_B3g_lbl)

  ! Equilibrium ASC in B3g symmetry.
  ! This is an internal check: the relevant segment of the vector
  ! should be the same as the one calculated using pcmsolver_compute_response_asc
  allocate (asc_B3g(grid_size))
  asc_B3g = 0.0_c_double
  ! This is the B3g irreducible representation
  call pcmsolver_compute_asc(pcm_context, &
                             mep_lbl, &
                             asc_B3g_lbl, &
                             irrep=3_c_int)
  call pcmsolver_get_surface_function(pcm_context, grid_size, asc_B3g, asc_B3g_lbl)

```

(continues on next page)

(continued from previous page)

```fortran
  ! Check that everything calculated is OK
  ! Cavity size
  if (grid_size .ne. ref_size) then
    write (error_unit, *) 'Error in the cavity size, please file an issue on: https://github.com/PCMSolver/pcmsolver'
    stop
  else
    write (output_unit, *) 'Test on cavity size: PASSED'
  end if
  ! Irreducible cavity size
  if (irr_grid_size .ne. ref_irr_size) then
    write (error_unit, *) 'Error in the irreducible cavity size, please file an issue on: https://github.com/PCMSolver/pcmsolver'
    stop
  else
    write (output_unit, *) 'Test on irreducible cavity size: PASSED'
  end if
  ! Polarization energy
  if (.not. check_unsigned_error(energy, ref_energy, 1.0e-7_c_double)) then
    write (error_unit, *) 'Error in the polarization energy, please file an issue on: https://github.com/PCMSolver/pcmsolver'
    stop
  else
    write (output_unit, *) 'Test on polarization energy: PASSED'
  end if
  ! Surface functions
  call test_surface_functions(grid_size, mep, asc_Ag, asc_B3g, asc_neq_B3g, areas)

  call pcmsolver_save_surface_function(pcm_context, mep_lbl)
  call pcmsolver_load_surface_function(pcm_context, mep_lbl)

  call pcmsolver_write_timings(pcm_context)

  call pcmsolver_delete(pcm_context)

  deallocate (charges)
  deallocate (coordinates)
  deallocate (grid)
  deallocate (mep)
  deallocate (asc_Ag)
  deallocate (asc_B3g)
  deallocate (asc_neq_B3g)
  deallocate (areas)

  close (output_unit)

end program pcm_fortran_host
```

1.6 Interfacing with a C host

Warning: Multidimensional arrays are handled in *column-major ordering* (i.e. Fortran ordering) by the module.

```
/*
 * PCMSolver, an API for the Polarizable Continuum Model
 * Copyright (C) 2016 Roberto Di Remigio, Luca Frediani and collaborators.
 *
 * This file is part of PCMSolver.
 *
 * PCMSolver is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Lesser General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * PCMSolver is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with PCMSolver.  If not, see <http://www.gnu.org/licenses/>.
 *
 * For information on the complete list of contributors to the
 * PCMSolver API, see: <http://pcmsolver.readthedocs.io/>
 */

#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "PCMInput.h"
#include "pcmsolver.h"

#include "C_host-functions.h"

#define NR_NUCLEI 6

FILE * output;

void host_writer(const char * message) { fprintf(output, "%s\n", message); }

int main() {

  output = fopen("C_host.out", "w+");
  if (!pcmsolver_is_compatible_library()) {
    fprintf(stderr, "%s\n", "PCMSolver library not compatible");
    exit(EXIT_FAILURE);
  }

  fprintf(output, "%s\n", "Starting a PCMSolver calculation");
  // Use C2H4 in D2h symmetry
  double charges[NR_NUCLEI] = {6.0, 1.0, 1.0, 6.0, 1.0, 1.0};
  double coordinates[3 * NR_NUCLEI] = {0.0,
```

(continues on next page)

(continued from previous page)

```
                                       0.000000,
                                       1.257892,
                                       0.0,
                                       1.745462,
                                       2.342716,
                                       0.0,
                                       -1.745462,
                                       2.342716,
                                       0.0,
                                       0.000000,
                                       -1.257892,
                                       0.0,
                                       1.745462,
                                       -2.342716,
                                       0.0,
                                       -1.745462,
                                       -2.342716};
  // This means the molecular point group has three generators:
  // the Oxy, Oxz and Oyz planes
  int symmetry_info[4] = {3, 4, 2, 1};
  struct PCMInput host_input = pcmsolver_input();

  pcmsolver_context_t * pcm_context = pcmsolver_new(PCMSOLVER_READER_HOST,
                                                    NR_NUCLEI,
                                                    charges,
                                                    coordinates,
                                                    symmetry_info,
                                                    &host_input,
                                                    host_writer);

  pcmsolver_citation(host_writer);

  pcmsolver_print(pcm_context);

  int grid_size = pcmsolver_get_cavity_size(pcm_context);
  int irr_grid_size = pcmsolver_get_irreducible_cavity_size(pcm_context);
  double * grid = (double *)calloc(3 * grid_size, sizeof(double));
  pcmsolver_get_centers(pcm_context, grid);
  double * areas = (double *)calloc(grid_size, sizeof(double));
  pcmsolver_get_areas(pcm_context, areas);

  double * mep = nuclear_mep(NR_NUCLEI, charges, coordinates, grid_size, grid);
  const char * mep_lbl = {"NucMEP"};
  pcmsolver_set_surface_function(pcm_context, grid_size, mep, mep_lbl);
  const char * asc_lbl = {"NucASC"};
  // This is the Ag irreducible representation (totally symmetric)
  int irrep = 0;
  pcmsolver_compute_asc(pcm_context, mep_lbl, asc_lbl, irrep);
  double * asc_Ag = (double *)calloc(grid_size, sizeof(double));
  pcmsolver_get_surface_function(pcm_context, grid_size, asc_Ag, asc_lbl);

  double energy =
      pcmsolver_compute_polarization_energy(pcm_context, mep_lbl, asc_lbl);

  fprintf(output, "Polarization energy: %20.12f\n", energy);

  double * asc_neq_B3g = (double *)calloc(grid_size, sizeof(double));
```

(continues on next page)

(continued from previous page)

```
  const char * asc_neq_B3g_lbl = {"OITASC"};
  // This is the B3g irreducible representation
  irrep = 3;
  pcmsolver_compute_response_asc(pcm_context, mep_lbl, asc_neq_B3g_lbl, irrep);
  pcmsolver_get_surface_function(
      pcm_context, grid_size, asc_neq_B3g, asc_neq_B3g_lbl);

  // Equilibrium ASC in B3g symmetry.
  // This is an internal check: the relevant segment of the vector
  // should be the same as the one calculated using pcmsolver_compute_response_asc
  double * asc_B3g = (double *)calloc(grid_size, sizeof(double));
  const char * asc_B3g_lbl = {"ASCB3g"};
  pcmsolver_compute_asc(pcm_context, mep_lbl, asc_B3g_lbl, irrep);
  pcmsolver_get_surface_function(pcm_context, grid_size, asc_B3g, asc_B3g_lbl);

  // Check that everything calculated is OK
  // Cavity size
  const int ref_size = 576;
  if (grid_size != ref_size) {
    fprintf(stderr,
            "%s\n",
            "Error in the cavity size, please file an issue on: "
            "https://github.com/PCMSolver/pcmsolver");
    exit(EXIT_FAILURE);
  } else {
    fprintf(output, "%s\n", "Test on cavity size: PASSED");
  }
  // Irreducible cavity size
  const int ref_irr_size = 72;
  if (irr_grid_size != ref_irr_size) {
    fprintf(stderr,
            "%s\n",
            "Error in the irreducible cavity size, please file an "
            "issue on: https://github.com/PCMSolver/pcmsolver");
    exit(EXIT_FAILURE);
  } else {
    fprintf(output, "%s\n", "Test on irreducible cavity size: PASSED");
  }
  // Polarization energy
  const double ref_energy = -0.437960027982;
  if (!check_unsigned_error(energy, ref_energy, 1.0e-7)) {
    fprintf(stderr,
            "%s\n",
            "Error in the polarization energy, please file an issue "
            "on: https://github.com/PCMSolver/pcmsolver");
    exit(EXIT_FAILURE);
  } else {
    fprintf(output, "%s\n", "Test on polarization energy: PASSED");
  }
  // Surface functions
  test_surface_functions(
      output, grid_size, mep, asc_Ag, asc_B3g, asc_neq_B3g, areas);

  pcmsolver_save_surface_functions(pcm_context);
  pcmsolver_save_surface_function(pcm_context, asc_lbl);
  pcmsolver_load_surface_function(pcm_context, mep_lbl);

```

(continues on next page)

(continued from previous page)

```
  pcmsolver_write_timings(pcm_context);

  pcmsolver_delete(pcm_context);

  free(grid);
  free(mep);
  free(asc_Ag);
  free(asc_B3g);
  free(asc_neq_B3g);
  free(areas);

  fclose(output);

  return EXIT_SUCCESS;
}
```

CHAPTER

TWO

PUBLICATIONS

2.1 Peer-reviewed journal articles

2.1.1 2015

- Four-Component Relativistic Calculations in Solution with the Polarizable Continuum Model of Solvation: Theory, Implementation, and Application to the Group 16 Dihydrides H2X (X = O, S, Se, Te, Po)
- Wavelet Formulation of the Polarizable Continuum Model. II. Use of Piecewise Bilinear Boundary Elements

2.1.2 2016

- A Polarizable Continuum Model for Molecules at Spherical Diffuse Interfaces

2.1.3 2017

- Four-Component Relativistic Density Functional Theory with the Polarizable Continuum Model: Application to EPR Parameters and Paramagnetic NMR Shifts
- Open-ended formulation of self-consistent field response theory with the polarizable continuum model for solvation
- Psi4 1.1: An Open-Source Electronic Structure Program Emphasizing Automation, Advanced Libraries, and Interoperability
- Combining frozen-density embedding with the conductor-like screening model using Lagrangian techniques for response properties

2.2 Theses

- The Polarizable Continuum Model Goes Viral! Extensible, Modular and Sustainable Development of Quantum Mechanical Continuum Solvation Models Doctoral thesis, Roberto Di Remigio, January 2017.

2.3 Presentations

- A modular implementation of the Polarizable Continuum Model for Solvation Presentation given by Roberto Di Remigio at the workshop in honour of professor Jacopo Tomasi's 80th birthday. Pisa, August 31 - September 1 2014.
- The Polarizable Continuum Model Goes Viral! PhD defense, Roberto Di Remigio, January 16 2017.
- PCMSolver: a modern, modular approach to include solvation in any quantum chemistry code. Presentation given by Luca Frediani at WATOC 2017. Munich, August 27 - September 1 2017.

2.4 Posters

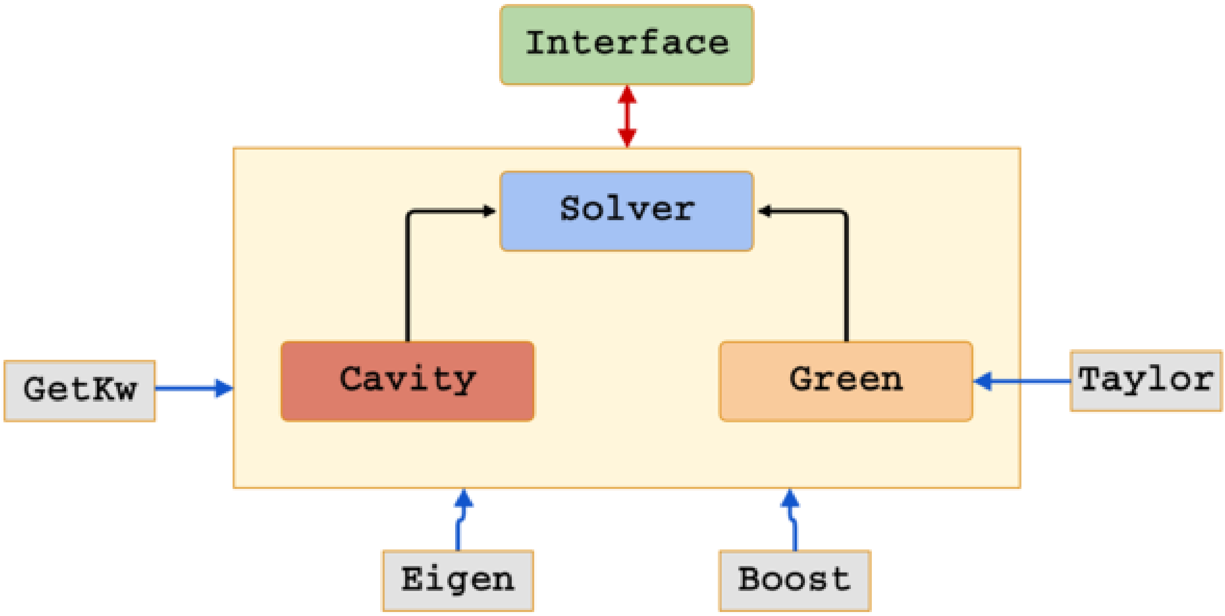
- Plug the solvent in your favorite QM program Presented by Luca Frediani at the 14th International Congress of Quantum Chemistry. Boulder, Colorado, June 25-30 2012.
- 4-Component Relativistic Calculations in Solution with the Polarizable Continuum Model of Solvation Presented by Roberto Di Remigio at the FemEx-Oslo conference. Oslo, June 13-16 2014.

CHAPTER

THREE

PCMSOLVER PROGRAMMERS' MANUAL

3.1 General Structure



External libraries:

- parts of the C++ Boost libraries are used to provide various functionality, like ordinary differential equations integrators. The source for the 1.54.0 release is shipped with the module's source code. Some of the libraries used need to be compiled. Boost is released under the terms of the Boost Software License, v1.0 (see also http://www.boost.org/users/license.html)

  > **Warning:** As of v1.1.11 we have started removing the dependency from Boost. The use of Boost is thus deprecated.

- the Eigen template library for linear algebra. Almost every operation involving matrices and vectors is performed through Eigen. Eigen provides convenient type definitions for vectors and matrices (of arbitrary dimensions) and the corresponding operations. Have a look here for a quick reference guide to the API and at the getting started guide to get started. Eigen is distributed under the terms of the Mozilla Public License, v2.0

- the Getkw library by Jonas Juselius is used to manage input. It is distributed under the terms of the GNU General Public License, v2.0
- the libtaylor library implementing automatic differentiation and available under the terms of the MIT License.

Third-party code snippets:

- Fortran subroutines *dsyevv3*, *dsyevh3*, *dsyevj3* for the diagonalization of 3x3 Hermitian matrices. These subroutines were copied verbatim from the source code provided by Joachim Kopp and described in [Kop08] (also available on the arXiv) The diagonalization subroutines are made available under the terms of the GNU Lesser General Public License, v2.1
- C++ cnpy library for saving arrays in C++ into Numpy arrays. The library is from Carl Rogers under the terms of the MIT License. The version in PCMSolver is slightly different.

3.2 Coding standards

General Object-Oriented design principles you should try to follow:

1. Identify the aspects of your application that vary and separate them from what stays the same;
2. Program to an interface, not an implementation;
3. Favor composition over inheritance;
4. Strive for loosely coupled designs between objects that interact;
5. Classes should be open for extension, but closed for modification;
6. Depend upon abstractions. Do not depend upon concrete classes;
7. Principle of Least Knowledge. Talk only to your immediate friends;

[SA04][CGL98][Cli]

3.2.1 Including header files

Do not include header files unnecessarily. Even if PCMSolver is not a big project, unnecessary include directives and/or forward declarations introduce nasty interdependencies among different parts of the code. This reflects mainly in longer compilation times, but also in uglier looking code (see also the discussion in [Sut99]).

Follow these guidelines to decide whether to include or forward declare:

1. class A makes no reference to class B. Neither include nor forward declare B;
2. class A refers to class B as a friend. Neither include nor forward declare B;
3. class A contains a pointer/reference to a class B object. Forward declare B;
4. class A contains functions with a class B object (value/pointer/reference) as parameter/return value. Forward declare B;
5. class A is derived from class B. include B;
6. class A contains a class B object. include B.

```
#pragma once

//====================================
// Forward declared dependencies
class Foo;
```

(continues on next page)

(continued from previous page)

```
class Bar;

//===============================
// Included dependencies
#include <vector>
#include "Parent.hpp"

//===============================
// The actual class
class MyClass : public Parent // Parent object, so #include "Parent.h"
{
  public:
    std::vector<int> avector; // vector object, so #include <vector>
    Foo * foo;                // Foo pointer, so forward declare
    void Func(Bar & bar);     // Bar reference as parameter, so forward declare

    friend class MyFriend;    // friend declaration is not a dependency
                              //    don't do anything about MyFriend
};
```

3.2.2 Proper overloading of *operator<<*

Suppose we have an inheritance hierarchy made of an abstract base class, Base, and two derived classes, Derived1 and Derived2. In the Base class header file we will define a pure virtual private function printObject and provide a public friend overload of operator<<:

```
#include <iosfwd>

class Base
{
  public:
    // All your other very fancy public members
    friend std::ostream & operator<<(std::ostream & os, Base & base)
    {
            return base.printObject(os);
    }
  protected:
    // All your other very fancy protected members
  private:
    // All your other very fancy private members
    virtual std::ostream & printObject(std::ostream & os) = 0;
}
```

The printObject method can also be made (impure) virtual, it really depends on your class hierarchy. Derived1 and Derived2 header files will provide a public friend overload of operator<< (friendliness isn't inherited, transitive or reciprocal) and an override for the printObject method:

```
#include <iosfwd>

#include "Base.hpp"

class Derived1 : public Base
{
  public:
    // All your other very fancy public members
```

(continues on next page)

(continued from previous page)

```
    friend std::ostream & operator<<(std::ostream & os, Derived1 & derived)
    {
      return derived.printObject(os);
    }
  protected:
    // All your other very fancy protected members
  private:
    // All your other very fancy private members
    virtual std::ostream & printObject(std::ostream & os);
}

class Derived2 : public Base
{
  public:
    // All your other very fancy public members
    friend std::ostream & operator<<(std::ostream & os, Derived2 & derived)
    {
      return derived.printObject(os);
    }
  protected:
    // All your other very fancy protected members
  private:
    // All your other very fancy private members
    virtual std::ostream & printObject(std::ostream & os);
}
```

3.2.3 Code formatting

We conform to the so-called Linux (aka kernel) formatting style for C/C++ code (see http://en.wikipedia.org/wiki/Indent_style#Kernel_style) with minimal modifications. Using clang-format is the preferred method to get the source code in the right format. Formatting style is defined in the `.clang-format` file, kept at the root of the project.

Note: We recommend using at least v3.9 of the program, which is the version used to generate the `.clang-format` file defining all formatting settings.

`clang-format` can be integrated with both Emacs and Vim. It is also possible to install the Git pre-commit hooks to perform the necessary code style checks prior to committing changes:

```
cd .git/hooks
cp --symbolic-link ../../.githooks/* .
```

3.3 Documentation

This documentation is generated using Sphinx and Doxygen The two softwares are bridged by means of the Breathe extension The online version of this documentation is built and served by Read The Docs. The webpage http://pcmsolver.readthedocs.org/ is updated on each push to the public GitHub repository.

3.3.1 How and what to document

Doxygen enables documenting the code in the source code files thus removing a “barrier” for developers. To avoid that the code degenerates into a Big Ball of Mud, it is mandatory to document directly within the source code classes and functions. To document general programming principles, design choices, maintenance etc. you can create a .rst file in the `doc` directory. Remember to refer the new file inside the `index.rst` file (it won’t be parsed otherwise). Sphing uses reStructuredText and Markdown. Support for Markdown is not as extensive as for reStructuredText, see these comments. Follow the guidelines in [WAB+14] regarding what to document.

Write the documentation in the header file. To document a class, put `/*! \class <myclass>` inside the namespace but before the class. Add the following to a `.rst` file:

```
.. doxygenclass:: <namespace>::<myclass>
 :project: PCMSolver
 :members:
 :protected-members:
 :private-members:
```

Do similar when documenting `struct`-s and complete files.

Note: Use `/*! */` to open and close a Doxygen comment.

3.3.2 Documenting methods in derived classes

Virtual methods should only be documented in the base classes. This avoids unnecessary verbosity and conforms to the principle: “Document _what_, not _how_” [WAB+14] If you feel the _how_ needs to be explicitly documented, add some notes in the appropriate `.rst` file.

3.3.3 How does this work?

To have an offline version of the documentation just issue in the `doc` folder:

```
sphinx-build . _build
```

The HTML will be stored in `_build/`. Open the `_build/index.html` file with your browser to see and browse the documentation.

Warning: It is only possible to build documentation locally from within the `doc` folder. This choice was made to simplify the set up of the ReadTheDocs and local documentation building procedures and to minimize the chances of breaking either.

Note: Building the documentation requires Python, Doxygen, Sphinx, Perl and the Python modules breathe, matplotlib, sphinx-rtd-theme, sphinxcontrib-bibtex and recommonmark. The required python modules can be installed by running `pip install -r requirements.txt`. There is also a `Pipfile` in case people prefer to use `pipenv`.

3.4 CMake usage

This is a brief guide to our CMake infrastructure which is managed *via* Autocmake

Warning: The minimum required CMake version is 2.8.10

3.4.1 Adding new source subdirectories and/or files

Developers **HAVE TO** manually list the sources in a given subdirectory of the main source directory `src/`. In our previous infrastructure this was not necessary, but the developers needed to trigger CMake to regenerate the Makefiles manually.

New subdirectory

First of all, you will have to let CMake know that a new source-containing subdirectory has been added to the source tree. Due to the hierarchical approach CMake is based upon you will need to modify the `CMakeLists.txt` in the `src` directory and create a new one in your new subdirectory. For the first step:

1. if your new subdirectory contains header files, add a line like the following to the `CMakeLists.txt` file contained in the `src` directory:

   ```
   ${CMAKE_CURRENT_LIST_DIR}/subdir_name
   ```

   to the command setting the list of directories containing headers. This sets up the list of directories where CMake will look for headers with definitions of classes and functions. If your directory contains Fortran code you can skip this step;

2. add a line like the following to the `CMakeLists.txt` file contained in the `src` directory:

   ```
   add_subdirectory(subdir_name)
   ```

   This will tell CMake to go look inside `subdir_name` for a `CMakeLists.txt` containing more sets of instructions. It is preferable to add these new lines in **alphabetic order**

Inside your new subdirectory you will need to add a `CMakeLists.txt` file containing the set of instructions to build your cutting edge code. This is the second step. Run the `make_cmake_files.py` Python script in the `src/` directory:

```
python make_cmake_files.py --libname=cavity --lang=CXX
```

to generate a template `CMakeLists.txt.try` file:

```
# List of headers
list(APPEND headers_list Cavity.hpp ICavity.hpp Element.hpp GePolCavity.hpp RegisterCavityToFactory.hpp RestartCavity.hpp)

# List of sources
list(APPEND sources_list ICavity.cpp Element.cpp GePolCavity.cpp RestartCavity.cpp)

add_library(cavity OBJECT ${sources_list} ${headers_list})
set_target_properties(cavity PROPERTIES POSITION_INDEPENDENT_CODE 1 )
set_property(GLOBAL APPEND PROPERTY PCMSolver_HEADER_DIRS ${CMAKE_CURRENT_LIST_DIR})
# Sets install directory for all the headers in the list
```

(continues on next page)

(continued from previous page)

```
foreach(_header ${headers_list})
   install(FILES ${_header} DESTINATION include/cavity)
endforeach()
```

The template might need additional editing. Each source subdirectory is the lowest possible in the CMake hierarchy and it contains set of instructions for:

1. exporting a list of header files (.h or .hpp) to the upper level in the hierarchy, possibly excluding some of them
2. define install targets for the files in this subdirectory.

All the source files are compiled into the unique static library `libpcm.a` and unique dynamic library `libpcm.so`. This library is the one the host QM program need to link.

Searching for libraries

In general, the use of the find_package macro is to be preferred, as it is standardized and ensured to work on any platform. Use of `find_package` requires that the package/library you want to use has already a module inside the CMake distribution. If that's not the case, you should *never* use the following construct for third-party libraries:

```
target_link_libraries(myexe -lsomesystemlib)
```

If the library does not exist, the end result is a cryptic linker error. See also Jussi Pakkanen's blog You will first need to find the library, using the macro find_library, and then use the `target_link_libraries` command.

3.5 Versioning and minting a new release

Our versioning machinery is based on a modified version of the `versioner.py` script devised by Lori A. Burns (Georgia Tech) for the Psi4 quantum chemistry code. The documentation that follows is also adapted from the corresponding Psi4 documentation, available at this link

This guide will walk you through the actions to perform to mint a new release of the code. Version numbering follows the guidelines of semantic versioning. The allowed format is `MAJOR.MINOR.PATCH-DESCRIBE`, where `DESCRIBE` can be a string describing a prerelease state, such as `rc2`, `alpha1`, `beta3` and so forth.

3.5.1 Minting a new release

The `tools/metadata.py` file records the versioning information for the current release. The information in this file is used by the `versioner.py` script to compute a *unique version number* for development snapshots.

Note: To correctly mint a new release, you will have to be on the latest release branch of (i) a direct clone or (ii) clone-of-fork with release branch up-to-date with upstream (including tags!!!) and with upstream as remote.

This is the step-by-step guide to releasing a new version of PCMSolver:

1. **DECIDE** an upcoming version number, say `1.2.0`.
2. **TIDY UP** `CHANGELOG.md`:
   - **SET** the topmost header to the upcoming version number and release date.

     ```
     ## [Version 1.2.0] - 2018-03-31
     ```

- **CHECK** that the links at the bottom of the document are correct.

  ```
  [Unreleased]: https://github.com/PCMSolver/pcmsolver/compare/v1.2.0...HEAD
  [Version 1.2.0]: https://github.com/PCMSolver/pcmsolver/compare/v1.2.0-rc1...
  ↪v1.2.0
  [Version 1.2.0-rc1]: https://github.com/PCMSolver/pcmsolver/compare/v1.1.12...
  ↪v1.2.0-rc1
  ```

3. **UPDATE** the `AUTHORS.md` file:

   - Run `git shortlog -sn` and cross-check with the current contents of `AUTHORS.md`. Edit where necessary and don't forget to include, where available, the GitHub handle. Authors are ordered by the number of commits.
   - Update the revision date at the bottom of this file.

   ```
   >>> cat AUTHORS.md
   ## Individual Contributors

   - Roberto Di Remigio (@robertodr)
   - Luca Frediani (@ilfreddy)
   - Monica Bugeanu (@mbugeanu)
   - Arnfinn Hykkerud Steindal (@arnfinn)
   - Radovan Bast (@bast)
   - T. Daniel Crawford (@lothian)
   - Krzysztof Mozgawa
   - Lori A. Burns (@loriab)
   - Ville Weijo (@vweijo)
   - Ward Poelmans (@wpoely86)

   This list was obtained 2018-03-02 by running `git shortlog -sn`
   ```

4. **CHECK** that the `.mailmap` file is up-to-date.
5. **CHECK** that the documentation builds locally.
6. **ACT** to check all the changed files in.
7. **OBSERVE** current versioning state

   - https://github.com/PCMSolver/pcmsolver/releases says `v1.2.0-rc1` & `9a8c391`

   ```
   >>> git tag
   v1.1.0
   v1.1.1
   v1.1.10
   v1.1.11
   v1.1.12
   v1.1.2
   v1.1.3
   v1.1.4
   v1.1.5
   v1.1.6
   v1.1.7
   v1.1.8
   v1.1.9
   v1.2.0-rc1

   >>> cat tools/metadata.py
   __version__ = '1.2.0-rc1'
   ```

(continues on next page)

(continued from previous page)

```
__version_long = '1.2.0-rc1+9a8c391'
__version_upcoming_annotated_v_tag = '1.2.0'
__version_most_recent_release = '1.1.12'


def version_formatter(dummy):
    return '(inplace)'

>>> git describe --abbrev=7 --long --always HEAD
v1.2.0-rc1-14-gfc02d9d

>>> git describe --abbrev=7 --long --dirty
v1.2.0-rc1-14-gfc02d9d-dirty

>>> python tools/versioner.py
Defining development snapshot version: 1.2.0.dev14+fc02d9d (computed)
1.2.0.dev14 {versioning-script} fc02d9d 1.1.12.999 dirty  1.1.12 <-- 1.2.
↪0.dev14+fc02d9d

>>> git diff
```

- Observe that current latest tag matches metadata script and git describe, that GH releases matches metadata script, that upcoming in metadata script matches current `versioner.py` version.

8. **ACT** to bump tag in code. The current tag is `v1.2.0-rc1`, the imminent tag is `v1.2.0`.

   - Edit current & prospective tag in `tools/metadata.py`. Use your decided-upon tag `v1.2.0` and a speculative next tag, say `v1.3.0`, and use 7 "z"s for the part you can't predict.

   ```
   >>> vim tools/metadata.py

   >>> git diff
   diff --git a/tools/metadata.py b/tools/metadata.py
   index 5d87b55..6cbc05e 100644
   --- a/tools/metadata.py
   +++ b/tools/metadata.py
   @@ -1,6 +1,6 @@
   -__version__ = '1.2.0-rc1'
   -__version_long = '1.2.0-rc1+9a8c391'
   -__version_upcoming_annotated_v_tag = '1.2.0'
   -__version_most_recent_release = '1.1.12'
   +__version__ = '1.2.0'
   +__version_long = '1.2.0+zzzzzzz'
   +__version_upcoming_annotated_v_tag = '1.3.0'
   +__version_most_recent_release = '1.2.0'
   ```

   - **COMMIT** changes to `tools/metadata.py`.

   ```
   >>> git add tools/metadata.py
   >>> git commit -m "Bump version to v1.2.0"
   ```

9. **OBSERVE** undefined version state. Note the 7-character git hash for the new commit, here `fc02d9d`.

```
>>> git describe --abbrev=7 --long --always HEAD
v1.2.0-rc1-14-gfc02d9d

>>> git describe --abbrev=7 --long --dirty
```

(continues on next page)

(continued from previous page)

```
v1.2.0-rc1-14-gfc02d9d-dirty

>>> python tools/versioner.py
Undefining version for irreconcilable tags: 1.2.0-rc1 (computed) vs 1.2.0␣
↪(recorded)
undefined {versioning-script} fc02d9d 1.2.0.999 dirty  1.2 <-- undefined+fc02d9d
```

10. **ACT** to bump tag in git, then bump git tag in code.

    - Use the decided-upon tag `v1.2.0` and the observed hash `fc02d9d` to mint a new *annotated* tag, minding that "v"s are present here.
    - Use the observed hash to edit `tools/metadata.py` and commit immediately.

```
>>> git tag -a v1.2.0 fc02d9d -m "Version 1.2.0 released"

>>> vim tools/metadata.py

>>> git diff
diff --git a/tools/metadata.py b/tools/metadata.py
index 6cbc05e..fdc202e 100644
--- a/tools/metadata.py
+++ b/tools/metadata.py
@@ -1,5 +1,5 @@
 __version__ = '1.2.0'
-__version_long = '1.2.0+zzzzzzz'
+__version_long = '1.2.0+fc02d9d'
 __version_upcoming_annotated_v_tag = '1.3.0'
 __version_most_recent_release = '1.2.0'

>>> python tools/versioner.py
Amazing, this can't actually happen that git hash stored at git commit.

>>> git add tools/metadata.py

>>> git commit -m "Records tag for v1.2.0"
```

11. **OBSERVE** current versioning state. There is nothing to take note of. This is just a snapshot to ensure that you did not mess up.

```
>>> python tools/versioner.py
Defining development snapshot version: 1.2.0.dev1+4e0596e (computed)
1.2.0.dev1 {master} 4e0596e 1.2.0.999   1.2 <-- 1.2.0.dev1+4e0596e

>>> git describe --abbrev=7 --long --always HEAD
v1.2.0-1-g4e0596e

>>> git describe --abbrev=7 --long --dirty
v1.2.0-1-g4e0596e

>>> git tag
v1.1.0
v1.1.1
v1.1.10
v1.1.11
v1.1.12
v1.1.2
```

(continues on next page)

(continued from previous page)

```
v1.1.3
v1.1.4
v1.1.5
v1.1.6
v1.1.7
v1.1.8
v1.1.9
v1.2.0-rc1
v1.2.0

>>> cat tools/metadata.py
__version__ = '1.2.0'
__version_long = '1.2.0+fc02d9d'
__version_upcoming_annotated_v_tag = '1.3.0'
__version_most_recent_release = '1.2.0'

>>> cat metadata.out.py | head -8
__version__ = '1.2.0.dev1'
__version_branch_name = 'master'
__version_cmake = '1.2.0.999'
__version_is_clean = 'True'
__version_last_release = '1.2.0'
__version_long = '1.2.0.dev1+4e0596e'
__version_prerelease = 'False'
__version_release = 'False'

>>> git log --oneline
4e0596e Records tag for v1.2.0
fc02d9d Bump version to v1.2.0
```

12. **ACT** to inform remote of bump

    - Temporarily disengage "Include administrators" on protected release branch.

    ```
    >>> git push origin release/1.2

    >>> git push origin v1.2.0
    ```

    - Now https://github.com/PCMSolver/pcmsolver/releases says `v1.2.0` & `fc023d9d`

13. **EDIT** release description in the GitHub web UI.

Zenodo will automatically generate a new, versioned DOI for the new release. It is no longer necessary to update the badge in the `README.md` since it will always resolve to the latest released by Zenodo.

3.5.2 How to create and remove an annotated Git tag on a remote

PCMSolver versioning only works with *annotated* tags, not *lightweight* tags as are created with the GitHub interface

- Create *annotated* tag:

```
>>> git tag -a v1.1.12 <git hash if not current> -m "Version 1.1.12 released"
>>> git push upstream --tags
```

- Delete tag:

```
>>> git tag -d v1.1.12
>>> git push origin :refs/tags/v1.1.12
```

- Pull tags:

```
>>> git fetch <remote> 'refs/tags/*:refs/tags/*'
```

3.6 Code contributions

We have adopted a fully public *fork and pull request* workflow, where every proposed changeset has to go through a code review and approval process.

The code changes are developed on a *branch* of the *fork*. When completed, the developer submits the changes for review through the web interface: a *pull request* (PR) is opened, requesting that the changes from the *source branch* on the fork be merged into a *target branch* in the canonical repository. Once the PR is open, the new code is automatically tested. Core developers of PCMSolver will then review the contribution and discuss additional changes to be made. Eventually, if all the tests are passing and a developer approves the suggested contribution, the changes are merged into the target branch. The target branch is (usually) the *master* branch, that is, the main development branch.

Note: All PRs goes to the master branch

The creator of the PR is responsible for keeping the code up to date with master, so the code in the PR reflects what will be the code in the master branch after merging.

3.6.1 Branching Model

We are using the stable mainline branching model for Git. In the main repository on github there are two types of branches:

- *one* main developing branch, called `master`
- release branches

A new release branch is created from the master branch for a new release, with the format `release/vMAJOR.MINOR`. A release branch will never be merged back to the master branch and will only receive bug fixes, thus no new features. These bug fixes would be cherry picked from the master branch, to ensure that the master branch always contains all bug fixes. In case a bug fix is only relevant for a given release, the bug should be fixed with a PR directly to the corresponding release branch. In case a bug fix is easy to perform on a release branch but challenging to perform on the master branch, the fix can be directed to a release branch. Then an issue *have* to be created to make sure it will also be fixed on the master branch.

Feature branches are not created on the main repository, but on forks. These are based on the master branch from the main repository and merged into the master branch through pull requests.

3.7 Changelog

We follow the guidelines of Keep a CHANGELOG On all **but** the release branches, there is an `Unreleased` section under which new additions should be listed. To simplify perusal of the `CHANGELOG.md`, use the following subsections:

1. `Added` for new features.
2. `Changed` for changes in existing functionality.
3. `Deprecated` for once-stable features removed in upcoming releases.
4. `Removed` for deprecated features removed in this release.
5. `Fixed` for any bug fixes.
6. `Security` to invite users to upgrade in case of vulnerabilities.

3.8 Updating Eigen Distribution

The C++ linear algebra library Eigen comes bundled with the module. To update the distributed version one has to:

1. download the desired version of the library to a scratch location. Eigen's website is: http://eigen.tuxfamily.org/
2. unpack the downloaded archive;
3. go into the newly created directory and create a build directory;
4. go into the newly created build directory and type the following (remember to substitute @PROJECT_SOURCE_DIR@ with the actual path)

   ```
   cmake .. -DCMAKE_INSTALL_PREFIX=@PROJECT_SOURCE_DIR@/external/eigen3
   ```

Remember to commit and push your modifications.

3.9 Git Pre-Commit Hooks

Git pre-commit hooks are used to keep track of code style and license header in source files. Code style is checked using `clang-format` for C/C++ and `yapf` for Python.

> **Warning: You need to install ``clang-format`` (v3.9 recommended) and ``yapf`` (v0.20 recommended) to run the code style validation hook!**

License headers are checked using the `license_maintainer.py` script and the header templates for the different languages used in this project. The Python script checks the `.gitattributes` file to determine which license headers need to be maintained and in which files:

```
src/pedra/pedra_dlapack.F90 !licensefile
src/solver/*.hpp licensefile=.githooks/LICENSE-C++
```

The first line specifies that the file in `src/pedra/pedra_dlapack.F90` should not be touched, while the second line states that all `.hpp` files in `src/solver` should get an header from the template in `.githooks/LICENSE-C++` Location of files in `.gitattributes` are always specified with respect to the project root directory.

The hooks are located in the `.githooks` subdirectory and **have to be installed by hand** whenever you clone the repository anew:

```
cd .git/hooks
cp --symbolic-link ../../.githooks/* .
```

Installed hooks will **always** be executed. Use `git commit --no-verify` to bypass explicitly the hooks.

3.10 Profiling

You should obtain profiling information before attempting any optimization of the code. There are many ways of obtaining this information, but we have only experimented with the following:

1. Using Linux `perf` and related tools.
2. Using `gperftools`.
3. Using Intel VTune.

Profiling should be done using the standalone executable `run_pcm` and any of the input files gathered under the `tests/benchmark` directory. These files are copied to the build directory. If you are lazy, you can run the profiling from the build directory:

```
>>> cd tests/benchmark

>>> env PYTHONPATH=<build_dir>/lib64/python:$PYTHONPATH
       python <build_dir>/bin/go_pcm.py --inp=standalone.pcm --exe=<build_dir>/bin
```

3.10.1 Using `perf`

`perf` is a tool available on Linux. Though part of the kernel tools, it is not usually preinstalled on most Linux distributions. For visualization purposes we also need additional tools, in particular the flame graph generation scripts Probably your distribution has them prepackaged already. `perf` will trace all CPU events on your system, hence you might need to fiddle with some kernel set up files to get permissions to trace events.

Note: `perf` **is NOT** available on `stallo`. Even if it were, you would probably not have permissions to record kernel traces.

These are the instructions I used:

1. Trace execution. This will save CPU stack traces to a `perf.data` file. Successive runs do not overwrite this file.

   ```
   >>> cd tests/benchmark

   >>> perf record -F 99 -g -- env PYTHONPATH=<build_dir>/lib64/python:$PYTHONPATH␣
    ↪python
                 <build_dir>/bin/go_pcm.py --inp=standalone.pcm --exe=<build_dir>/bin
   ```

2. Get reports. There are different ways of getting a report from the `perf.data` file. The following will generate a call tree.

   ```
   >>> perf report --stdio
   ```

3. Generate an interactive flame graph.

```
>>> perf script | stackcollapse-perf.pl > out.perf-folded

>>> cat out.perf-folded | flamegraph.pl > perf-run_pcm.svg
```

3.10.2 Using gperftools

This set of tools was previously known as Google Performance Tools. The executable needs to be linked against the `profiler`, `tcmalloc` and `unwind` libraries. CMake will attempt to find them. If this fails, you will have to install them, you should either check if they are available for your distribution or compile from source. In principle, one could use the `LD_PRELOAD` mechanism to skip the *ad hoc* compilation of the executable.

Note: `gperftools` **is** available on `stallo`, but it's an ancient version.

1. Configure the code with the `--gperf` option enabled. CPU and heap profiling, together with heap-checking will be available.

2. CPU profiling can be done with the following command:

   ```
   >>> env CPUPROFILE=run_pcm.cpu.prof PYTHONPATH=<build_dir>/lib64/python:
   ↪$PYTHONPATH
           python <build_dir>/bin/go_pcm.py --inp=standalone.pcm --exe=<build_dir>/
   ↪bin
   ```

   This will save the data to the `run_pcm.cpu.prof` file. To analyze the gathered data we can use the `pprof` script:

   ```
   >>> pprof --text <build_dir>/bin/run_pcm run_pcm.cpu.prof
   ```

   This will print a table. Any row will look like the following:

   ```
   2228   7.2%  24.8%    28872  93.4% pcm::utils::splineInterpolation
   ```

   where the columns respectively report:

   1. Number of profiling samples in this function.
   2. Percentage of profiling samples in this function.
   3. Percentage of profiling samples in the functions printed so far.
   4. Number of profiling samples in this function and its callees.
   5. Percentage of profiling samples in this function and its callees.
   6. Function name.

   For more details look here

3. Heap profiling can be done with the following command:

   ```
   >>> env HEAPPROFILE=run_pcm.hprof PYTHONPATH=<build_dir>/lib64/python:$PYTHONPATH
           python <build_dir>/bin/go_pcm.py --inp=standalone.pcm --exe=<build_dir>/
   ↪bin
   ```

This will output a series of datafiles run_pcm.hprof.0000.heap, run_pcm.hprof.0001.heap and so forth. You will have to kill execution when enough samples have been collected. Analysis of the heap profiling data can be done using pprof. Read more here

3.10.3 Using Intel VTune

This is probably the easiest way to profile the code. VTune is Intel software, it might be possible to get a personal, free license. The instructions will hold on any machine where VTune is installed and you can look for more details on the online documentation You can, in principle, use the GUI. I haven't managed to do that though.

On stallo, start an interactive job and load the following modules:

```
>>> module load intel/2018a

>>> module load CMake

>>> module load VTune

>>> export BOOST_INCLUDEDIR=/home/roberto/Software/boost/include

>>> export BOOST_LIBRARYDIR=/home/roberto/Software/boost/lib
```

You will need to compile with optimizations activated, *i.e.* release mode. It is better to first parse the input file and then call run_pcm:

```
>>> cd <build_dir>/tests/benchmark

>>> env PYTHONPATH=../../lib64/python:$PYTHONPATH
    python ../../bin/go_pcm.py --inp=standalone_bubble.pcm
```

To start collecting hotspots:

```
>>> amplxe-cl -collect hotspots ../../bin/run_pcm @standalone_bubble.pcm
```

VTune will generate a folder r000hs with the collected results. A report for the hotspots can be generated with:

```
>>> amplxe-cl -report hotspots -r r000hs > report
```

3.11 Testing

We perform unit testing of our API. The unit testing framework used is Catch The framework provides quite an extensive set of macros to test various data types, it also provides facilities for easily setting up test fixtures. Usage is extremely simple and the documentation is very well written. For a quick primer on how to use Catch refer to: https://github.com/philsquared/Catch/blob/master/docs/tutorial.md The basic idea of unit testing is to test each building block of the code separataly. In our case, the term "building block" is used to mean a class.

To add new tests for your class you have to:

1. create a new subdirectory inside tests/ and add a line like the following to the CMakeLists.txt

   ```
   add_subdirectory(new_subdir)
   ```

2. create a CMakeLists.txt inside your new subdirectory. This CMakeLists.txt adds the source for a given unit test to the global UnitTestsSources property and notifies CTest that a test with given name is

part of the test suite. The generation of the `CMakeLists.txt` can be managed by `make_cmake_files.py` Python script. This will take care of also setting up CTest labels. This helps in further grouping the tests for our convenience. Catch uses tags to index tests and tags are surrounded by square brackets. The Python script inspects the sources and extracts labels from Catch tags. The `add_Catch_test` CMake macro takes care of the rest:

```
add_Catch_test(
  NAME
    <test-name> # Mandatory!
  LABELS
    <test-labels> # Mandatory! One per line, for readability
  DEPENDS
    <test-dependencies> # Optional. One per line, for readability
  REFERENCE_FILES
    <test-refs> # Optional. One per line, for readability
  COST
    <test-cost> # Optional. Roughly the seconds it takes to run the test
)
```

We require that each source file containing tests follows the naming convention new_subdir_testname and that testname gives some clue to what is being tested. Depending on the execution of tests in a different subdirectory is bad practice. A possible workaround is to add some kind of input file and create a text fixture that sets up the test environment. Have a look in the `tests/input` directory for an example

3. create the `.cpp` files containing the tests. Use the following template:

```
/*
 * PCMSolver, an API for the Polarizable Continuum Model
 * Copyright (C) 2016 Roberto Di Remigio, Luca Frediani and collaborators.
 *
 * This file is part of PCMSolver.
 *
 * PCMSolver is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Lesser General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * PCMSolver is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU Lesser General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with PCMSolver.  If not, see <http://www.gnu.org/licenses/>.
 *
 * For information on the complete list of contributors to the
 * PCMSolver API, see: <http://pcmsolver.readthedocs.io/>
 */

#include "catch.hpp"

#include <cmath>
#include <vector>

#include <Eigen/Core>

#include "TestingMolecules.hpp"
#include "cavity/GePolCavity.hpp"
```

(continues on next page)

(continued from previous page)

```

SCENARIO("GePol cavity for a single sphere", "[gepol][gepol_point]") {
  GIVEN("A single sphere") {
    double area = 0.4;
    double probeRadius = 0.0;
    double minRadius = 100.0;
    WHEN("the sphere is obtained from a Molecule object") {
      Molecule point = dummy<0>();
      GePolCavity cavity = GePolCavity(point, area, probeRadius, minRadius, "point
↪");
      cavity.saveCavity("point.npz");

      /*! \class GePolCavity
       *  \test \b GePolCavityTest_size tests GePol cavity size for a point␣
↪charge in
       * C1 symmetry without added spheres
       */
      THEN("the size of the cavity is") {
        int size = 32;
        int actualSize = cavity.size();
        REQUIRE(size == actualSize);
      }
      /*! \class GePolCavity
       *  \test \b GePolCavityTest_area tests GePol cavity surface area for a␣
↪point
       * charge in C1 symmetry without added spheres
       */
      AND_THEN("the surface area of the cavity is") {
        double area = 4.0 * M_PI * pow(1.0, 2);
        double actualArea = cavity.elementArea().sum();
        REQUIRE(area == Approx(actualArea));
      }
      /*! \class GePolCavity
       *  \test \b GePolCavityTest_volume tests GePol cavity volume for a point
       * charge in C1 symmetry without added spheres
       */
      AND_THEN("the volume of the cavity is") {
        double volume = 4.0 * M_PI * pow(1.0, 3) / 3.0;
        Eigen::Matrix3Xd elementCenter = cavity.elementCenter();
        Eigen::Matrix3Xd elementNormal = cavity.elementNormal();
        double actualVolume = 0;
        for (int i = 0; i < cavity.size(); ++i) {
          actualVolume +=
              cavity.elementArea(i) * elementCenter.col(i).dot(elementNormal.
↪col(i));
        }
        actualVolume /= 3;
        REQUIRE(volume == Approx(actualVolume));
      }
    }
  }

  GIVEN("A single sphere") {
    double area = 0.4;
    double probeRadius = 0.0;
    double minRadius = 100.0;
    WHEN("the sphere is obtained from a Sphere object") {
```

(continues on next page)

(continued from previous page)

```
      Sphere sph(Eigen::Vector3d::Zero(), 1.0);
      GePolCavity cavity = GePolCavity(sph, area, probeRadius, minRadius, "point");

      /*! \class GePolCavity
       *  \test \b GePolCavitySphereCTORTest_size tests GePol cavity size for a point
       * charge in C1 symmetry without added spheres
       */
      THEN("the size of the cavity is") {
        int size = 32;
        int actualSize = cavity.size();
        REQUIRE(size == actualSize);
      }
      /*! \class GePolCavity
       *  \test \b GePolCavitySphereCTORTest_area tests GePol cavity surface area for
       * a point charge in C1 symmetry without added spheres
       */
      AND_THEN("the surface area of the cavity is") {
        double area = 4.0 * M_PI * pow(1.0, 2);
        double actualArea = cavity.elementArea().sum();
        REQUIRE(area == Approx(actualArea));
      }
      /*! \class GePolCavity
       *  \test \b GePolCavitySphereCTORTest_volume tests GePol cavity volume for a
       * point charge in C1 symmetry without added spheres
       */
      AND_THEN("the volume of the cavity is") {
        double volume = 4.0 * M_PI * pow(1.0, 3) / 3.0;
        Eigen::Matrix3Xd elementCenter = cavity.elementCenter();
        Eigen::Matrix3Xd elementNormal = cavity.elementNormal();
        double actualVolume = 0;
        for (int i = 0; i < cavity.size(); ++i) {
          actualVolume +=
              cavity.elementArea(i) * elementCenter.col(i).dot(elementNormal.col(i));
        }
        actualVolume /= 3;
        REQUIRE(volume == Approx(actualVolume));
      }
    }
  }
}
```

In this example we are creating a test fixture. The fixture will instatiate a `GePolCavity` with fixed parameters. The result is then tested against reference values in the various `SECTION` s. It is **important** to add the documentation lines on top of the tests, to help other developers understand which class is being tested and what parameters are being tested. Within Catch fixtures are created behind the curtains, you do not need to worry about those details. This results in somewhat terser test source files.

3.12 Timer class

The `Timer` class enables timing of execution throughout the module. Timer support is enabled by passing `-DENABLE_TIMER=ON` to the `setup.py` script. Timing macros are available by inclusion of the `Config.hpp` header file.

The class is basically a wrapper around an ordered map of strings and cpu timers. To time a code snippet:

```
TIMER_ON("code-snippet");
// code-snippet
TIMER_OFF("code-snippet");
```

The timings are printed out to the `pcmsolver.timer.dat` by a call to the `TIMER_DONE` macro. This should obviously happen at the very end of the execution!

Defines

`TIMER_ON``(...)`

`TIMER_OFF``(...)`

`TIMER_DONE``(...)`

CHAPTER

FOUR

CLASSES AND FUNCTIONS REFERENCE

4.1 Cavities

We will here describe the inheritance hierarchy for generating cavities, in order to use and extend it properly. The runtime creation of cavity objects relies on the Factory Method pattern [GHJV94][Ale01], implemented through the generic Factory class.

4.1.1 ICavity

class *pcm*::**ICavity**

Abstract Base Class for cavities.

This class represents a cavity made of spheres, its surface being discretized in terms of finite elements.

Author Krzysztof Mozgawa

Date 2011

Subclassed by *pcm::cavity::GePolCavity*, *pcm::cavity::RestartCavity*

Public Functions

ICavity()

Default constructor.

ICavity(**const** Sphere &*sph*)

Constructor from a single sphere.

Only used when we have to deal with a single sphere, i.e. in the unit tests

Parameters

- `[in] sph`: the sphere

ICavity(**const** std::vector<Sphere> &*sph*)

Constructor from list of spheres.

Only used when we have to deal with a single sphere, i.e. in the unit tests

Parameters

- `[in] sph`: the list of spheres

`ICavity`(**`const`** *Molecule* &*molec*)

Constructor from *Molecule*.

Parameters

- `[in] molec`: the molecular aggregate

void **`saveCavity`**(**`const`** std::string &*fname* = "cavity.npz")

Save cavity specification to file.

The cavity specification contains: 0. the number of finite elements, nElements;

i. the weight of the finite elements, elementArea;

ii. the radius of the finite elements, elementRadius;

iii. the centers of the finite elements, elementCenter;

iv. the normal vectors relative to the centers, elementNormal. Each of these objects is saved in a separate .npy binary file and compressed into one .npz file. Notice that this is just the minimal set of data needed to restart an energy calculation.

void **`loadCavity`**(**`const`** std::string &*fname* = "cavity.npz")

Load cavity specification from file.

Protected Attributes

std::vector<Sphere> **`spheres_`**

List of spheres.

Molecule **`molecule_`**

The molecule to be wrapped by the cavity.

PCMSolverIndex **`nElements_`**

Number of finite elements generated.

PCMSolverIndex **`nIrrElements_`**

Number of irreducible finite elements.

bool **`built`**

Whether the cavity has been built.

Eigen::Matrix3Xd **`elementCenter_`**

Coordinates of elements centers.

Eigen::Matrix3Xd **`elementNormal_`**

Outward-pointing normal vectors to the elements centers.

Eigen::VectorXd **`elementArea_`**

Elements areas.

int **`nSpheres_`**

Number of spheres.

Eigen::Matrix3Xd **`elementSphereCenter_`**

Centers of the sphere the elements belong to.

Eigen::VectorXd **`elementRadius_`**

Radii of the sphere the elements belong to.

Eigen::Matrix3Xd **sphereCenter_**
Spheres centers.

Eigen::VectorXd **sphereRadius_**
Spheres radii.

std::vector<Element> **elements_**
List of finite elements.

Symmetry **pointGroup_**
Molecular point group.

Private Functions

void **makeCavity**() = 0
Creates the cavity and discretizes its surface.

Has to be implemented by classes lower down in the inheritance hierarchy

4.1.2 GePolCavity

class *pcm*::cavity::**GePolCavity** : **public** *pcm*::*ICavity*
A class for GePol cavity.

This class is an interface to the Fortran code PEDRA for the generation of cavities according to the GePol algorithm.

Author Krzysztof Mozgawa, Roberto Di Remigio

Date 2011, 2016

Private Functions

void **makeCavity**() **override**
Creates the cavity and discretizes its surface.

Has to be implemented by classes lower down in the inheritance hierarchy

void **build**(**const** std::string &*suffix*, int *maxts*, int *maxsp*, int *maxvert*)
Driver for PEDRA Fortran module.

Parameters

- `[in] suffix`: for the cavity.off and PEDRA.OUT files, the PID will also be added
- `[in] maxts`: maximum number of tesserae
- `[in] maxsp`: maximum number of spheres (original + added)
- `[in] maxvert`: maximum number of vertices

void **writeOFF**(**const** std::string &*suffix*)
Writes the cavity.off file for visualizing the cavity.

Parameters

- `[in] suffix`: for the cavity.off The full name of the visualization file will be cavity.off_suffix_PID

4.1.3 RestartCavity

class *pcm*::cavity::**RestartCavity** : **public** *pcm*::*ICavity*

A class for Restart cavity.

Author Roberto Di Remigio

Date 2014

Public Functions

void **makeCavity**() **override**

Creates the cavity and discretizes its surface.

Has to be implemented by classes lower down in the inheritance hierarchy

4.2 Green's Functions

We will here describe the inheritance hierarchy for generating Green's functions, in order to use and extend it properly. The runtime creation of Green's functions objects relies on the Factory Method pattern [GHJV94][Ale01], implemented through the generic Factory class.

The top-level header, _i.e._ to be included in client code, is `Green.hpp`. The common interface to all Green's function classes is specified by the `IGreensFunction` class, this is non-templated. All other classes are templated. The Green's functions are registered to the factory based on a label encoding: type, derivative, and dielectric profile. The only allowed labels must be listed in `src/green/Green.hpp`. If they are not, they can not be selected at run time.

4.2.1 IGreensFunction

class *pcm*::**IGreensFunction**

Interface for Green's function classes.

We **define** as *Green's function* a function:

$$G(\mathbf{r}, \mathbf{r}') : \mathbb{R}^6 \rightarrow \mathbb{R}$$

Green's functions and their directional derivatives appear as kernels of the $\mathcal{S}$ and $\mathcal{D}$ integral operators. Forming the matrix representation of these operators requires performing integrations over surface finite elements. Since these Green's functions present a Coulombic divergence, the diagonal elements of the operators will diverge unless appropriately formulated. This is possible, but requires **explicit** access to the *subtype* of this abstract base object. This justifies the need for the singleLayer and doubleLayer functions. The code uses the Non-Virtual Interface (NVI) idiom.

Author Luca Frediani and Roberto Di Remigio

Date 2012-2016

Subclassed by *pcm::green::GreensFunction< DerivativeTraits, dielectric_profile::Anisotropic >*, *pcm::green::GreensFunction< DerivativeTraits, dielectric_profile::Sharp >*, *pcm::green::GreensFunction< DerivativeTraits, dielectric_profile::Uniform >*, *pcm::green::GreensFunction< DerivativeTraits, dielectric_profile::Yukawa >*, *pcm::green::GreensFunction< DerivativeTraits, ProfilePolicy >*, pcm::green::GreensFunction< Stencil, ProfilePolicy >

Unnamed Group

double **kernelS**(**const** Eigen::Vector3d &*p1*, **const** Eigen::Vector3d &*p2*) **const**

Methods to sample the Green's function and its probe point directional derivative

Returns value of the kernel of the $\mathcal{S}$ integral operator, i.e. the value of the Greens's function for the pair of points p1, p2: $G(\mathbf{p}_1, \mathbf{p}_2)$

Note This is the Non-Virtual Interface (NVI)

Parameters

- `[in] p1`: first point
- `[in] p2`: second point

double **kernelD**(**const** Eigen::Vector3d &*direction*, **const** Eigen::Vector3d &*p1*, **const** Eigen::Vector3d &*p2*) **const**

Returns value of the kernel of the $\mathcal{D}$ integral operator for the pair of points p1, p2: $[\varepsilon\nabla_{\mathbf{p_2}}G(\mathbf{p}_1, \mathbf{p}_2)]\cdot\mathbf{n}_{\mathbf{p}_2}$ To obtain the kernel of the $\mathcal{D}^\dagger$ operator call this methods with $\mathbf{p}_1$ and $\mathbf{p}_2$ exchanged and with $\mathbf{n}_{\mathbf{p}_2} = \mathbf{n}_{\mathbf{p}_1}$

Note This is the Non-Virtual Interface (NVI)

Parameters

- `[in] direction`: the direction
- `[in] p1`: first point
- `[in] p2`: second point

Unnamed Group

double **singleLayer**(**const** Element &*e*, double *factor*) **const**

Methods to compute the diagonal of the matrix representation of the S and D operators by approximate collocation.

Calculates an element on the diagonal of the matrix representation of the S operator using an approximate collocation formula.

Note This is the Non-Virtual Interface (NVI)

Parameters

- `[in] e`: finite element on the cavity
- `[in] factor`: the scaling factor for the diagonal elements

double **doubleLayer**(**const** Element &*e*, double *factor*) **const**

Calculates an element of the diagonal of the matrix representation of the D operator using an approximate collocation formula.

Note This is the Non-Virtual Interface (NVI)

Parameters

- `[in] e`: finite element on the cavity
- `[in] factor`: the scaling factor for the diagonal elements

Unnamed Group

double **kernelS_impl**(**const** Eigen::Vector3d &*p1*, **const** Eigen::Vector3d &*p2*) **const** = 0

Methods to sample the Green's function and its probe point directional derivative

Returns value of the kernel of the $\mathcal{S}$ integral operator, i.e. the value of the Greens's function for the pair of points p1, p2: $G(\mathbf{p}_1, \mathbf{p}_2)$

Parameters

- `[in] p1`: first point
- `[in] p2`: second point

double **kernelD_impl**(**const** Eigen::Vector3d &*direction*, **const** Eigen::Vector3d &*p1*, **const** Eigen::Vector3d &*p2*) **const** = 0

Returns value of the kernel of the $\mathcal{D}$ integral operator for the pair of points p1, p2: $[\varepsilon\nabla_{\mathbf{p_2}}G(\mathbf{p}_1, \mathbf{p}_2)] \cdot \mathbf{n}_{\mathbf{p}_2}$ To obtain the kernel of the $\mathcal{D}^\dagger$ operator call this methods with $\mathbf{p}_1$ and $\mathbf{p}_2$ exchanged and with $\mathbf{n}_{\mathbf{p}_2} = \mathbf{n}_{\mathbf{p}_1}$

Parameters

- `[in] direction`: the direction
- `[in] p1`: first point
- `[in] p2`: second point

Unnamed Group

double **singleLayer_impl**(**const** Element &*e*, double *factor*) **const** = 0

Methods to compute the diagonal of the matrix representation of the S and D operators by approximate collocation.

Calculates an element on the diagonal of the matrix representation of the S operator using an approximate collocation formula.

Parameters

- `[in] e`: finite element on the cavity
- `[in] factor`: the scaling factor for the diagonal elements

double **doubleLayer_impl**(**const** Element &*e*, double *factor*) **const** = 0

Calculates an element of the diagonal of the matrix representation of the D operator using an approximate collocation formula.

Parameters

- `[in] e`: finite element on the cavity
- `[in] factor`: the scaling factor for the diagonal elements

Public Functions

bool **uniform**() **const** = 0
Whether the Green's function describes a uniform environment

double **permittivity**() **const** = 0
Returns a dielectric permittivity

4.2.2 GreensFunction

template<typename **DerivativeTraits**, typename **ProfilePolicy**>
class *pcm*::green::**GreensFunction** : **public** *pcm*::*IGreensFunction*
Templated interface for Green's functions.

Author Luca Frediani and Roberto Di Remigio

Date 2012-2016

Template Parameters

- `DerivativeTraits`: evaluation strategy for the function and its derivatives
- `ProfilePolicy`: dielectric profile type

Unnamed Group

double **derivativeSource**(**const** Eigen::Vector3d *&normal_p1*, **const** Eigen::Vector3d *&p1*, **const** Eigen::Vector3d *&p2*) **const**
Methods to sample the Green's function directional derivatives

Returns value of the directional derivative of the Greens's function for the pair of points p1, p2: $\nabla_{\mathbf{p_1}}G(\mathbf{p}_1, \mathbf{p}_2) \cdot \mathbf{n}_{\mathbf{p}_1}$ Notice that this method returns the directional derivative with respect to the source point.

Parameters

- `[in] normal_p1`: the normal vector to p1
- `[in] p1`: first point
- `[in] p2`: second point

double **derivativeProbe**(**const** Eigen::Vector3d *&normal_p2*, **const** Eigen::Vector3d *&p1*, **const** Eigen::Vector3d *&p2*) **const final**
Returns value of the directional derivative of the Greens's function for the pair of points p1, p2: $\nabla_{\mathbf{p_2}}G(\mathbf{p}_1, \mathbf{p}_2) \cdot \mathbf{n}_{\mathbf{p}_2}$ Notice that this method returns the directional derivative with respect to the probe point.

Parameters

- `[in] normal_p2`: the normal vector to p2
- `[in] p1`: first point
- `[in] p2`: second point

Unnamed Group

Eigen::Vector3d **gradientSource**(**const** Eigen::Vector3d &*p1*, **const** Eigen::Vector3d &*p2*) **const**

Methods to sample the Green's function gradients

Returns full gradient of Greens's function for the pair of points p1, p2: $\nabla_{\mathbf{p_1}}G(\mathbf{p}_1, \mathbf{p}_2)$ Notice that this method returns the gradient with respect to the source point.

Parameters

- `[in] p1`: first point
- `[in] p2`: second point

Eigen::Vector3d **gradientProbe**(**const** Eigen::Vector3d &*p1*, **const** Eigen::Vector3d &*p2*) **const**

Returns full gradient of Greens's function for the pair of points p1, p2: $\nabla_{\mathbf{p_2}}G(\mathbf{p}_1, \mathbf{p}_2)$ Notice that this method returns the gradient with respect to the probe point.

Parameters

- `[in] p1`: first point
- `[in] p2`: second point

Public Functions

bool **uniform**() **const final override**

Whether the Green's function describes a uniform environment

Protected Functions

DerivativeTraits **operator()**(*DerivativeTraits* **source*, *DerivativeTraits* **probe*) **const** = 0

Evaluates the Green's function given a pair of points

Parameters

- `[in] source`: the source point
- `[in] probe`: the probe point

double **kernelS_impl**(**const** Eigen::Vector3d &*p1*, **const** Eigen::Vector3d &*p2*) **const final override**

Returns value of the kernel of the $\mathcal{S}$ integral operator, i.e. the value of the Greens's function for the pair of points p1, p2: $G(\mathbf{p}_1, \mathbf{p}_2)$

Note Relies on the implementation of operator() in the subclasses and that is all subclasses need to implement. Thus this method is marked final.

Parameters

- `[in] p1`: first point
- `[in] p2`: second point

Protected Attributes

double **delta_**

Step for numerical differentiation.

ProfilePolicy **profile_**

Permittivity profile.

4.2.3 Vacuum

template<typename **DerivativeTraits** = AD_directional>
class *pcm*::green::**Vacuum** : **public** *pcm*::green::*GreensFunction*<*DerivativeTraits*, dielectric_profile::*Uniform*>

Green's function for vacuum.

Author Luca Frediani and Roberto Di Remigio

Date 2012-2016

Template Parameters

- DerivativeTraits: evaluation strategy for the function and its derivatives

Public Functions

double **permittivity**() **const final override**

Returns a dielectric permittivity

Private Functions

DerivativeTraits **operator()**(*DerivativeTraits* **sp*, *DerivativeTraits* **pp*) **const override**

Evaluates the Green's function given a pair of points

Parameters

- [in] source: the source point
- [in] probe: the probe point

double **kernelD_impl**(**const** Eigen::Vector3d &*direction*, **const** Eigen::Vector3d &*p1*, **const** Eigen::Vector3d &*p2*) **const override**

Returns value of the kernel of the $\mathcal{D}$ integral operator for the pair of points p1, p2: $[\varepsilon\nabla_{\mathbf{p_2}}G(\mathbf{p}_1, \mathbf{p}_2)] \cdot \mathbf{n}_{\mathbf{p}_2}$ To obtain the kernel of the $\mathcal{D}^\dagger$ operator call this methods with $\mathbf{p}_1$ and $\mathbf{p}_2$ exchanged and with $\mathbf{n}_{\mathbf{p}_2} = \mathbf{n}_{\mathbf{p}_1}$

Parameters

- [in] direction: the direction
- [in] p1: first point
- [in] p2: second point

double **singleLayer_impl**(**const** Element &*e*, double *factor*) **const override**

Methods to compute the diagonal of the matrix representation of the S and D operators by approximate collocation.

Calculates an element on the diagonal of the matrix representation of the S operator using an approximate collocation formula.

Parameters

- `[in] e`: finite element on the cavity
- `[in] factor`: the scaling factor for the diagonal elements

double **`doubleLayer_impl`**(**`const`** Element &*e*, double *factor*) **`const override`**

Calculates an element of the diagonal of the matrix representation of the D operator using an approximate collocation formula.

Parameters

- `[in] e`: finite element on the cavity
- `[in] factor`: the scaling factor for the diagonal elements

4.2.4 UniformDielectric

template<typename **`DerivativeTraits`** = AD_directional>
`class` *pcm*::`green`::**`UniformDielectric`** : **`public`** *pcm*::green::*GreensFunction*<*DerivativeTraits*, dielectric_profile::*Unif*

Green’s function for uniform dielectric.

Author Luca Frediani and Roberto Di Remigio

Date 2012-2016

Template Parameters

- `DerivativeTraits`: evaluation strategy for the function and its derivatives

Public Functions

double **`permittivity`**() **`const final override`**

Returns a dielectric permittivity

Private Functions

DerivativeTraits **`operator()`**(*DerivativeTraits* **sp*, *DerivativeTraits* **pp*) **`const override`**

Evaluates the Green’s function given a pair of points

Parameters

- `[in] source`: the source point
- `[in] probe`: the probe point

double **`kernelD_impl`**(**`const`** Eigen::Vector3d &*direction*, **`const`** Eigen::Vector3d &*p1*, **`const`** Eigen::Vector3d &*p2*) **`const override`**

Returns value of the kernel of the $\mathcal{D}$ integral operator for the pair of points p1, p2: $[\varepsilon\nabla_{\mathbf{p_2}}G(\mathbf{p}_1, \mathbf{p}_2)]\cdot\mathbf{n}_{\mathbf{p}_2}$ To obtain the kernel of the $\mathcal{D}^\dagger$ operator call this methods with $\mathbf{p}_1$ and $\mathbf{p}_2$ exchanged and with $\mathbf{n}_{\mathbf{p}_2} = \mathbf{n}_{\mathbf{p}_1}$

Parameters

- `[in] direction`: the direction
- `[in] p1`: first point
- `[in] p2`: second point

double **singleLayer_impl**(**const** Element &*e*, double *factor*) **const override**

Methods to compute the diagonal of the matrix representation of the S and D operators by approximate collocation.

Calculates an element on the diagonal of the matrix representation of the S operator using an approximate collocation formula.

Parameters

- `[in] e`: finite element on the cavity
- `[in] factor`: the scaling factor for the diagonal elements

double **doubleLayer_impl**(**const** Element &*e*, double *factor*) **const override**

Calculates an element of the diagonal of the matrix representation of the D operator using an approximate collocation formula.

Parameters

- `[in] e`: finite element on the cavity
- `[in] factor`: the scaling factor for the diagonal elements

4.2.5 IonicLiquid

template<typename **DerivativeTraits** = AD_directional>
class *pcm*::green::**IonicLiquid** : **public** *pcm*::green::*GreensFunction*<*DerivativeTraits*, dielectric_profile::*Yukawa*>

Green's functions for ionic liquid, described by the linearized Poisson-Boltzmann equation.

Author Luca Frediani, Roberto Di Remigio

Date 2013-2016

Template Parameters

- `DerivativeTraits`: evaluation strategy for the function and its derivatives

Public Functions

double **permittivity**() **const final override**

Returns a dielectric permittivity

Private Functions

DerivativeTraits **operator()**(*DerivativeTraits* **sp*, *DerivativeTraits* **pp*) **const override**

Evaluates the Green's function given a pair of points

Parameters

- `[in] source`: the source point
- `[in] probe`: the probe point

double **kernelD_impl**(**const** Eigen::Vector3d &*direction*, **const** Eigen::Vector3d &*p1*, **const** Eigen::Vector3d &*p2*) **const override**

Returns value of the kernel of the $\mathcal{D}$ integral operator for the pair of points p1, p2: $[\varepsilon\nabla_{\mathbf{p_2}}G(\mathbf{p}_1, \mathbf{p}_2)] \cdot \mathbf{n}_{\mathbf{p}_2}$ To obtain the kernel of the $\mathcal{D}^\dagger$ operator call this methods with $\mathbf{p}_1$ and $\mathbf{p}_2$ exchanged and with $\mathbf{n}_{\mathbf{p}_2} = \mathbf{n}_{\mathbf{p}_1}$

Parameters

- `[in] direction`: the direction
- `[in] p1`: first point
- `[in] p2`: second point

double **singleLayer_impl**(**const** Element&, double) **const override**

Methods to compute the diagonal of the matrix representation of the S and D operators by approximate collocation.

Calculates an element on the diagonal of the matrix representation of the S operator using an approximate collocation formula.

Parameters

- `[in] e`: finite element on the cavity
- `[in] factor`: the scaling factor for the diagonal elements

double **doubleLayer_impl**(**const** Element&, double) **const override**

Calculates an element of the diagonal of the matrix representation of the D operator using an approximate collocation formula.

Parameters

- `[in] e`: finite element on the cavity
- `[in] factor`: the scaling factor for the diagonal elements

4.2.6 AnisotropicLiquid

template<typename **DerivativeTraits** = AD_directional>
class *pcm*::green::**AnisotropicLiquid** : **public** *pcm*::green::*GreensFunction*<*DerivativeTraits*, dielectric_profile::*Aniso*

Green's functions for anisotropic liquid, described by a tensorial permittivity.

Author Roberto Di Remigio

Date 2016

Template Parameters

- `DerivativeTraits`: evaluation strategy for the function and its derivatives

Public Functions

AnisotropicLiquid(**const** Eigen::Vector3d &*eigen_eps*, **const** Eigen::Vector3d &*euler_ang*)

Parameters

- `[in] eigen_eps`: eigenvalues of the permittivity tensors
- `[in] euler_ang`: Euler angles in degrees

double **permittivity**() **const final override**

Returns a dielectric permittivity

Private Functions

DerivativeTraits **operator()** (*DerivativeTraits* **sp*, *DerivativeTraits* **pp*) **const override**

Evaluates the Green's function given a pair of points

Parameters

- `[in] source`: the source point
- `[in] probe`: the probe point

double **kernelD_impl** (**const** Eigen::Vector3d &*direction*, **const** Eigen::Vector3d &*p1*, **const** Eigen::Vector3d &*p2*) **const override**

Returns value of the kernel of the $\mathcal{D}$ integral operator for the pair of points p1, p2: $[\varepsilon\nabla_{\mathbf{p_2}}G(\mathbf{p}_1, \mathbf{p}_2)] \cdot \mathbf{n}_{\mathbf{p}_2}$ To obtain the kernel of the $\mathcal{D}^\dagger$ operator call this methods with $\mathbf{p}_1$ and $\mathbf{p}_2$ exchanged and with $\mathbf{n}_{\mathbf{p}_2} = \mathbf{n}_{\mathbf{p}_1}$

Parameters

- `[in] direction`: the direction
- `[in] p1`: first point
- `[in] p2`: second point

double **singleLayer_impl** (**const** Element&, double) **const override**

Methods to compute the diagonal of the matrix representation of the S and D operators by approximate collocation.

Calculates an element on the diagonal of the matrix representation of the S operator using an approximate collocation formula.

Parameters

- `[in] e`: finite element on the cavity
- `[in] factor`: the scaling factor for the diagonal elements

double **doubleLayer_impl** (**const** Element&, double) **const override**

Calculates an element of the diagonal of the matrix representation of the D operator using an approximate collocation formula.

Parameters

- `[in] e`: finite element on the cavity
- `[in] factor`: the scaling factor for the diagonal elements

4.2.7 SphericalDiffuse

template<typename **ProfilePolicy** = dielectric_profile::*OneLayerLog*>
class *pcm*::green::**SphericalDiffuse** : **public** *pcm*::green::*GreensFunction*<Stencil, *ProfilePolicy*>

Green's function for a diffuse interface with spherical symmetry.

The origin of the dielectric sphere can be changed by means of the constructor. The solution of the differential equation defining the Green's function is **always** performed assuming that the dielectric sphere is centered in the origin of the coordinate system. Whenever the public methods are invoked to "sample" the Green's function at a pair of points, a translation of the sampling points is performed first.

Author Hui Cao, Ville Weijo, Luca Frediani and Roberto Di Remigio

Date 2010-2015

Template Parameters

- `ProfilePolicy`: functional form of the diffuse layer

Unnamed Group

int **`maxLGreen_`**

Parameters and functions for the calculation of the Green's function, including Coulomb singularity

Maximum angular momentum in the final summation over Legendre polynomials to obtain G

std::vector<RadialFunction<detail::StateType, detail::LnTransformedRadial, Zeta>> **`zeta_`**

First independent radial solution, used to build Green's function.

Note The vector has dimension maxLGreen_ and has r^l behavior

std::vector<RadialFunction<detail::StateType, detail::LnTransformedRadial, Omega>> **`omega_`**

Second independent radial solution, used to build Green's function.

Note The vector has dimension maxLGreen_ and has r^(-l-1) behavior

double **`imagePotentialComponent_impl`**(int *L*, **`const`** Eigen::Vector3d &*sp*, **`const`** Eigen::Vector3d &*pp*, double *Cr12*) **`const`**

Returns L-th component of the radial part of the Green's function.

Note This function shifts the given source and probe points by the location of the dielectric sphere.

Parameters

- `[in] L`: angular momentum
- `[in] sp`: source point
- `[in] pp`: probe point
- `[in] Cr12`: Coulomb singularity separation coefficient

Unnamed Group

int **`maxLC_`**

Parameters and functions for the calculation of the Coulomb singularity separation coefficient

Maximum angular momentum to obtain C(r, r'), needed to separate the Coulomb singularity

RadialFunction<detail::StateType, detail::LnTransformedRadial, Zeta> **`zetaC_`**

First independent radial solution, used to build coefficient.

Note This is needed to separate the Coulomb singularity and has r^l behavior

RadialFunction<detail::StateType, detail::LnTransformedRadial, Omega> **`omegaC_`**

Second independent radial solution, used to build coefficient.

Note This is needed to separate the Coulomb singularity and has r^(-l-1) behavior

double **`coefficient_impl`**(**`const`** Eigen::Vector3d &*sp*, **`const`** Eigen::Vector3d &*pp*) **`const`**

Returns coefficient for the separation of the Coulomb singularity.

Note This function shifts the given source and probe points by the location of the dielectric sphere.

Parameters

- `[in] sp`: first point
- `[in] pp`: second point

Public Functions

`SphericalDiffuse`(double *e1*, double *e2*, double *w*, double *c*, **const** Eigen::Vector3d &*o*, int *l*)
Constructor for a one-layer interface

Parameters

- `[in] e1`: left-side dielectric constant
- `[in] e2`: right-side dielectric constant
- `[in] w`: width of the interface layer
- `[in] c`: center of the diffuse layer
- `[in] o`: center of the sphere
- `[in] l`: maximum value of angular momentum

double **`permittivity`**() **const final override**
Returns a dielectric permittivity

double **`coefficientCoulomb`**(**const** Eigen::Vector3d &*source*, **const** Eigen::Vector3d &*probe*) **const**
Returns Coulomb singularity separation coefficient.

Parameters

- `[in] source`: location of the source charge
- `[in] probe`: location of the probe charge

double **`Coulomb`**(**const** Eigen::Vector3d &*source*, **const** Eigen::Vector3d &*probe*) **const**
Returns singular part of the Green's function.

Parameters

- `[in] source`: location of the source charge
- `[in] probe`: location of the probe charge

double **`imagePotential`**(**const** Eigen::Vector3d &*source*, **const** Eigen::Vector3d &*probe*) **const**
Returns non-singular part of the Green's function (image potential)

Parameters

- `[in] source`: location of the source charge
- `[in] probe`: location of the probe charge

double **`coefficientCoulombDerivative`**(**const** Eigen::Vector3d &*direction*, **const** Eigen::Vector3d &*p1*, **const** Eigen::Vector3d &*p2*) **const**
Returns value of the directional derivative of the Coulomb singularity separation coefficient for the pair of points p1, p2: $\nabla_{\mathbf{p_2}}G(\mathbf{p}_1, \mathbf{p}_2)\cdot *\mathbf{n}_{\mathbf{p}_2}$ Notice that this method returns the directional derivative with respect to the probe point, thus assuming that the direction is relative to that point.

Parameters

- `[in] direction`: the direction
- `[in] p1`: first point
- `[in] p2`: second point

double **`CoulombDerivative`**(**`const`** Eigen::Vector3d *&direction*, **`const`** Eigen::Vector3d *&p1*, **`const`** Eigen::Vector3d *&p2*) **`const`**

Returns value of the directional derivative of the singular part of the Greens's function for the pair of points p1, p2: $\nabla_{\mathbf{p_2}}G(\mathbf{p}_1, \mathbf{p}_2)\cdot *\mathbf{n}_{\mathbf{p}_2}$ Notice that this method returns the directional derivative with respect to the probe point, thus assuming that the direction is relative to that point.

Parameters

- `[in] direction`: the direction
- `[in] p1`: first point
- `[in] p2`: second point

double **`imagePotentialDerivative`**(**`const`** Eigen::Vector3d *&direction*, **`const`** Eigen::Vector3d *&p1*, **`const`** Eigen::Vector3d *&p2*) **`const`**

Returns value of the directional derivative of the non-singular part (image potential) of the Greens's function for the pair of points p1, p2: $\nabla_{\mathbf{p_2}}G(\mathbf{p}_1, \mathbf{p}_2)\cdot *\mathbf{n}_{\mathbf{p}_2}$ Notice that this method returns the directional derivative with respect to the probe point, thus assuming that the direction is relative to that point.

Parameters

- `[in] direction`: the direction
- `[in] p1`: first point
- `[in] p2`: second point

std::tuple<double, double> **`epsilon`**(**`const`** Eigen::Vector3d *&point*) **`const`**

Handle to the dielectric profile evaluation

Private Functions

Stencil **`operator()`**(Stencil **sp*, Stencil **pp*) **`const override`**

Evaluates the Green's function given a pair of points

Note This takes care of the origin shift

Parameters

- `[in] sp`: the source point
- `[in] pp`: the probe point

double **`kernelD_impl`**(**`const`** Eigen::Vector3d *&direction*, **`const`** Eigen::Vector3d *&p1*, **`const`** Eigen::Vector3d *&p2*) **`const override`**

Returns value of the kernel of the $\mathcal{D}$ integral operator for the pair of points p1, p2: $[\varepsilon\nabla_{\mathbf{p_2}}G(\mathbf{p}_1, \mathbf{p}_2)]\cdot\mathbf{n}_{\mathbf{p}_2}$ To obtain the kernel of the $\mathcal{D}^\dagger$ operator call this methods with $\mathbf{p}_1$ and $\mathbf{p}_2$ exchanged and with $\mathbf{n}_{\mathbf{p}_2} = \mathbf{n}_{\mathbf{p}_1}$

Parameters

- `[in] direction`: the direction

- `[in] p1`: first point
- `[in] p2`: second point

double **`singleLayer_impl`**`(`**`const`** Element &*e*, double *factor*`)` **`const override`**

Methods to compute the diagonal of the matrix representation of the S and D operators by approximate collocation.

Calculates an element on the diagonal of the matrix representation of the S operator using an approximate collocation formula.

Parameters

- `[in] e`: finite element on the cavity
- `[in] factor`: the scaling factor for the diagonal elements

double **`doubleLayer_impl`**`(`**`const`** Element &*e*, double *factor*`)` **`const override`**

Calculates an element of the diagonal of the matrix representation of the D operator using an approximate collocation formula.

Parameters

- `[in] e`: finite element on the cavity
- `[in] factor`: the scaling factor for the diagonal elements

void **`initSphericalDiffuse`**`()`

This calculates all the components needed to evaluate the Green's function

Private Members

Eigen::Vector3d **`origin_`**

Center of the dielectric sphere

4.3 Dielectric profiles

4.3.1 Uniform

`struct Uniform`

a uniform dielectric profile

Author Roberto Di Remigio

Date 2015

4.3.2 Anisotropic

class *pcm*::dielectric_profile::**Anisotropic**

describes a medium with anisotropy, i.e. liquid crystal

Author Roberto Di Remigio

Date 2014

Public Functions

Anisotropic(**const** Eigen::Vector3d &*eigen_eps*, **const** Eigen::Vector3d &*euler_ang*)

Parameters

- `[in] eigen_eps`: eigenvalues of the permittivity tensors
- `[in] euler_ang`: Euler angles in degrees

Private Functions

void **build**()

Initializes some internals: molecule-fixed to lab-fixed frame rotation matrix, permittivity tensor in molecule-fixed frame and its inverse

Private Members

Eigen::Vector3d **epsilonLab_**

Diagonal of the permittivity tensor in the lab-fixed frame.

Eigen::Vector3d **eulerAngles_**

Euler angles (in degrees) relating molecule-fixed and lab-fixed frames.

Eigen::Matrix3d **epsilon_**

Permittivity tensor in molecule-fixed frame.

Eigen::Matrix3d **epsilonInv_**

Inverse of the permittivity tensor in molecule-fixed frame.

Eigen::Matrix3d **R_**

molecule-fixed to lab-fixed frames rotation matrix

double **detEps_**

Determinant of the permittivity tensor.

4.3.3 Yukawa

struct Yukawa

describes a medium with damping, i.e. ionic liquid

Author Roberto Di Remigio

Date 2015

4.3.4 OneLayerLog

class *pcm*::dielectric_profile::**OneLayerLog**

A dielectric profile based on the Harrison and Fosso-Tande work [3].

Author Luca Frediani

Date 2017

Public Functions

std::tuple<double, double> **operator()** (**const** double *r*) **const**

Returns a tuple holding the permittivity and its derivative

Parameters

- `[in] r`: evaluation point

Private Functions

double **value** (double *point*) **const**

Returns value of dielectric profile at given point

Parameters

- `[in] point`: where to evaluate the profile

double **derivative** (double *point*) **const**

Returns value of derivative of dielectric profile at given point

Parameters

- `[in] point`: where to evaluate the derivative

Private Members

double **epsilon1_**

Dielectric constant on the left of the interface.

double **epsilon2_**

Dielectric constant one the right of the interface.

double **width_**

Width of the transition layer.

double **center_**

Center of the transition layer.

std::pair<double, double> **domain_**

Domain of the permittivity function This is formally $[0, +\infty)$, for all practical purposes the permittivity function is equal to the epsilon2_ already at 6.0 * width_ Thus the upper limit in the domain_ is initialized as center_ + 12.0 * width_

4.3.5 OneLayerTanh

class *pcm*::dielectric_profile::**OneLayerTanh**

A tanh dielectric profile as in [4].

Author Roberto Di Remigio

Date 2014

Note The parameter given from user input for width_ is divided by 6.0 in the constructor to keep consistency with [4]

Public Functions

std::tuple<double, double> **operator()**(**const** double *r*) **const**

Returns a tuple holding the permittivity and its derivative

Parameters

- `[in] r`: evaluation point

Private Functions

double **value**(double *point*) **const**

Returns value of dielectric profile at given point

Note We return epsilon2_ when the sampling point is outside the upper limit.

Parameters

- `[in] point`: where to evaluate the profile

double **derivative**(double *point*) **const**

Returns value of derivative of dielectric profile at given point

Note We return 0.0 (derivative of the constant value epsilon2_) when the sampling point is outside the upper limit.

Parameters

- `[in] point`: where to evaluate the derivative

Private Members

double **epsilon1_**

Dielectric constant on the left of the interface.

double **epsilon2_**

Dielectric constant one the right of the interface.

double **width_**

Width of the transition layer.

double **center_**

Center of the transition layer.

std::pair<double, double> **domain_**

Domain of the permittivity function This is formally $[0, +\infty)$, for all practical purposes the permittivity function is equal to the epsilon2_ already at 6.0 * width_ Thus the upper limit in the domain_ is initialized as center_ + 12.0 * width_

4.3.6 OneLayerErf

class *pcm*::dielectric_profile::**OneLayerErf**

A erf dielectric profile.

Author Roberto Di Remigio

Date 2015

Note The parameter given from user input for width_ is divided by 6.0 in the constructor to keep consistency with [4]

Public Functions

std::tuple<double, double> **operator()** (**const** double *r*) **const**

Returns a tuple holding the permittivity and its derivative

Parameters

- `[in] r`: evaluation point

Private Functions

double **value** (double *point*) **const**

Returns value of dielectric profile at given point

Note We return epsilon2_ when the sampling point is outside the upper limit.

Parameters

- `[in] point`: where to evaluate the profile

double **derivative** (double *point*) **const**

Returns value of derivative of dielectric profile at given point

Note We return 0.0 (derivative of the constant value epsilon2_) when the sampling point is outside the upper limit.

Parameters

- `[in] point`: where to evaluate the derivative

Private Members

double **epsilon1_**
Dielectric constant on the left of the interface.

double **epsilon2_**
Dielectric constant one the right of the interface.

double **width_**
Width of the transition layer.

double **center_**
Center of the transition layer.

std::pair<double, double> **domain_**
Domain of the permittivity function This is formally $[0, +\infty)$, for all practical purposes the permittivity function is equal to the epsilon2_ already at 6.0 * width_ Thus the upper limit in the domain_ is initialized as center_ + 12.0 * width_

4.3.7 Sharp

struct Sharp
A sharp dielectric separation.

Author Roberto Di Remigio

Date 2015

4.4 Solvers

We will here describe the inheritance hierarchy for generating solvers, in order to use and extend it properly. The runtime creation of solver objects relies on the Factory Method pattern [GHJV94][Ale01], implemented through the generic Factory class.

4.4.1 ISolver

class *pcm*::**ISolver**
Abstract Base Class for solvers inheritance hierarchy.

We use the Non-Virtual Interface idiom.

Author Luca Frediani, Roberto Di Remigio

Date 2011, 2015, 2016

Subclassed by *pcm::solver::CPCMSolver*, *pcm::solver::IEFSolver*

Public Functions

void **buildSystemMatrix**(**const** *ICavity* &*cavity*, **const** *IGreensFunction* &*gf_i*, **const** *IGreensFunction* &*gf_o*, **const** *IBoundaryIntegralOperator* &*op*)

Calculation of the PCM matrix.

Parameters

- [in] cavity: the cavity to be used
- [in] gf_i: Green's function inside the cavity
- [in] gf_o: Green's function outside the cavity
- [in] op: integrator strategy for the single and double layer operators

Eigen::VectorXd **computeCharge**(**const** Eigen::VectorXd &*potential*, int *irrep* = 0) **const**

Returns the ASC given the MEP and the desired irreducible representation.

Parameters

- [in] potential: the vector containing the MEP at cavity points
- [in] irrep: the irreducible representation of the MEP and ASC

Protected Functions

void **buildSystemMatrix_impl**(**const** *ICavity* &*cavity*, **const** *IGreensFunction* &*gf_i*, **const** *IGreensFunction* &*gf_o*, **const** *IBoundaryIntegralOperator* &*op*) = 0

Calculation of the PCM matrix.

Parameters

- [in] cavity: the cavity to be used
- [in] gf_i: Green's function inside the cavity
- [in] gf_o: Green's function outside the cavity
- [in] op: integrator strategy for the single and double layer operators

Eigen::VectorXd **computeCharge_impl**(**const** Eigen::VectorXd &*potential*, int *irrep* = 0) **const** = 0

Returns the ASC given the MEP and the desired irreducible representation.

Parameters

- [in] potential: the vector containing the MEP at cavity points
- [in] irrep: the irreducible representation of the MEP and ASC

Protected Attributes

bool **built_**
Whether the system matrix has been built

bool **isotropic_**
Whether the solver is isotropic

4.4.2 IEFSolver

class *pcm*::solver::**IEFSolver** : **public** *pcm*::*ISolver*
IEFPCM, collocation-based solver.

Author Luca Frediani, Roberto Di Remigio

Date 2011, 2015, 2016

Note We store the non-Hermitian, symmetry-blocked T(epsilon) and Rinfinity matrices. The ASC is obtained by multiplying the MEP by Rinfinity and then using a partially pivoted LU decomposition of T(epsilon) on the resulting vector. In case the polarization weights are requested, we use the approach suggested in [2]. First, the adjoint problem is solved:

$$\mathbf{T}_\varepsilon^\dagger \tilde{v} = v$$

Also in this case a partially pivoted LU decomposition is used. The "transposed" ASC is obtained by the matrix-vector multiplication:

$$q^* = \mathbf{R}_\infty^\dagger \tilde{v}$$

Eventually, the two sets of charges are summed and divided by 2 This avoids computing and storing the inverse explicitly, at the expense of storing both T(epsilon) and Rinfinity.

Public Functions

IEFSolver(bool *symm*)
Construct solver.

Parameters

- [in] symm: whether the system matrix has to be symmetrized

void **buildAnisotropicMatrix**(**const** *ICavity* &*cavity*, **const** *IGreensFunction* &*gf_i*, **const** *IGreensFunction* &*gf_o*, **const** *IBoundaryIntegralOperator* &*op*)
Builds PCM matrix for an anisotropic environment.

Parameters

- [in] cavity: the cavity to be used.
- [in] gf_i: Green's function inside the cavity
- [in] gf_o: Green's function outside the cavity
- [in] op: integrator strategy for the single and double layer operators

void **buildIsotropicMatrix**(**const** *ICavity* &*cavity*, **const** *IGreensFunction* &*gf_i*, **const** *IGreensFunction* &*gf_o*, **const** *IBoundaryIntegralOperator* &*op*)

Builds PCM matrix for an isotropic environment.

Parameters

- `[in] cavity`: the cavity to be used.
- `[in] gf_i`: Green's function inside the cavity
- `[in] gf_o`: Green's function outside the cavity
- `[in] op`: integrator strategy for the single and double layer operators

Private Functions

void **buildSystemMatrix_impl**(**const** *ICavity* &*cavity*, **const** *IGreensFunction* &*gf_i*, **const** *IGreensFunction* &*gf_o*, **const** *IBoundaryIntegralOperator* &*op*) **override**

Calculation of the PCM matrix.

Parameters

- `[in] cavity`: the cavity to be used
- `[in] gf_i`: Green's function inside the cavity
- `[in] gf_o`: Green's function outside the cavity
- `[in] op`: integrator strategy for the single and double layer operators

Eigen::VectorXd **computeCharge_impl**(**const** Eigen::VectorXd &*potential*, int *irrep* = 0) **const override**

Returns the ASC given the MEP and the desired irreducible representation.

Parameters

- `[in] potential`: the vector containing the MEP at cavity points
- `[in] irrep`: the irreducible representation of the MEP and ASC

Private Members

bool **hermitivitize_**

Whether the system matrix has to be symmetrized

Eigen::MatrixXd **Tepsilon_**

T(epsilon) matrix, not symmetry blocked

std::vector<Eigen::MatrixXd> **blockTepsilon_**

T(epsilon) matrix, symmetry blocked form

Eigen::MatrixXd **Rinfinity_**

R_infinity matrix, not symmetry blocked

std::vector<Eigen::MatrixXd> **blockRinfinity_**

R_infinity matrix, symmetry blocked form

4.4.3 CPCMSolver

class *pcm*::solver::**CPCMSolver** : **public** *pcm*::*ISolver*

Solver for conductor-like approximation: C-PCM (COSMO)

Author Roberto Di Remigio

Date 2013, 2016

Note We store the scaled, Hermitian, symmetrized S matrix and use a robust Cholesky decomposition to solve for the ASC. This avoids computing and storing the inverse explicitly. The S matrix is already scaled by the dielectric factor entering the definition of the conductor model!

Public Functions

CPCMSolver(bool *symm*, double *corr*)

Construct solver.

Parameters

- `[in] symm`: whether the system matrix has to be symmetrized
- `[in] corr`: factor to correct the conductor results

Private Functions

void **buildSystemMatrix_impl**(**const** *ICavity* &*cavity*, **const** *IGreensFunction* &*gf_i*, **const** *IGreensFunction* &*gf_o*, **const** *IBoundaryIntegralOperator* &*op*) **override**

Calculation of the PCM matrix.

Parameters

- `[in] cavity`: the cavity to be used
- `[in] gf_i`: Green's function inside the cavity
- `[in] gf_o`: Green's function outside the cavity
- `[in] op`: integrator strategy for the single layer operator

Eigen::VectorXd **computeCharge_impl**(**const** Eigen::VectorXd &*potential*, int *irrep* = 0) **const override**

Returns the ASC given the MEP and the desired irreducible representation.

Parameters

- `[in] potential`: the vector containing the MEP at cavity points
- `[in] irrep`: the irreducible representation of the MEP and ASC

Private Members

bool **hermitivitize_**
Whether the system matrix has to be symmetrized

double **correction_**
Correction for the conductor results

Eigen::MatrixXd **S_**
S matrix, not symmetry blocked

std::vector<Eigen::MatrixXd> **blockS_**
S matrix, symmetry blocked form

4.5 Boundary integral operators

4.5.1 IBoundaryIntegralOperator

class *pcm*::**IBoundaryIntegralOperator**
Subclassed by *pcm::bi_operators::Collocation*, *pcm::bi_operators::Numerical*, *pcm::bi_operators::Purisima*

Public Functions

Eigen::MatrixXd **computeS**(**const** *ICavity* &*cav*, **const** *IGreensFunction* &*gf*) **const**
Computes the matrix representation of the single layer operator

Parameters

- `[in] cav`: the discretized cavity
- `[in] gf`: a Green's function

Eigen::MatrixXd **computeD**(**const** *ICavity* &*cav*, **const** *IGreensFunction* &*gf*) **const**
Computes the matrix representation of the double layer operator

Parameters

- `[in] cav`: the discretized cavity
- `[in] gf`: a Green's function

Private Functions

Eigen::MatrixXd **computeS_impl**(**const** std::vector<cavity::Element> &*elems*, **const** *IGreensFunction* &*gf*) **const** = 0
Computes the matrix representation of the single layer operator

Parameters

- `[in] elems`: list of finite elements of the discretized cavity
- `[in] gf`: a Green's function

Eigen::MatrixXd **computeD_impl**(**const** std::vector<cavity::Element> *&elems*, **const** *IGreensFunction* *&gf*) **const** = 0

Computes the matrix representation of the double layer operator

Parameters

- `[in] elems`: list of finite elements of the discretized cavity
- `[in] gf`: a Green's function

4.5.2 Collocation

class *pcm*::bi_operators::**Collocation** : **public** *pcm*::*IBoundaryIntegralOperator*

Implementation of the single and double layer operators matrix representation using one-point collocation.

Calculates the diagonal elements of S as:

$$S_{ii} = factor * \sqrt{\frac{4\pi}{a_i}}$$

while the diagonal elements of D are:

$$D_{ii} = -factor * \sqrt{\frac{\pi}{a_i}} \frac{1}{R_I}$$

Author Roberto Di Remigio

Date 2015, 2016

Private Functions

Eigen::MatrixXd **computeS_impl**(**const** std::vector<cavity::Element> *&elems*, **const** *IGreensFunction* *&gf*) **const override**

Computes the matrix representation of the single layer operator

Parameters

- `[in] elems`: list of finite elements of the discretized cavity
- `[in] gf`: a Green's function

Eigen::MatrixXd **computeD_impl**(**const** std::vector<cavity::Element> *&elems*, **const** *IGreensFunction* *&gf*) **const override**

Computes the matrix representation of the double layer operator

Parameters

- `[in] elems`: list of finite elements of the discretized cavity
- `[in] gf`: a Green's function

Private Members

double **factor_**

Scaling factor for the diagonal elements of the matrix representation of the S and D operators

4.5.3 Purisima

class *pcm*::bi_operators::**Purisima** : **public** *pcm*::*IBoundaryIntegralOperator*

Implementation of the double layer operator matrix representation using one-point collocation and *Purisima*'s strategy for the diagonal of D.

Calculates the diagonal elements of D as:

$$D_{ii} = -\left(2\pi + \sum_{j\neq i} D_{ij} a_j \right) \frac{1}{a_i}$$

The original reference is [5]

Author Roberto Di Remigio

Date 2015, 2016

Private Functions

Eigen::MatrixXd **computeS_impl**(**const** std::vector<cavity::Element> &*elems*, **const** *IGreensFunction* &*gf*) **const override**

Computes the matrix representation of the single layer operator

Parameters

- [in] elems: list of finite elements of the discretized cavity
- [in] gf: a Green's function

Eigen::MatrixXd **computeD_impl**(**const** std::vector<cavity::Element> &*elems*, **const** *IGreensFunction* &*gf*) **const override**

Computes the matrix representation of the double layer operator by collocation using the *Purisima* sum rule to compute the diagonal elements. The sum rule for the diagonal elements is:

$$D_{ii} = -\left(2\pi + \sum_{j\neq i} D_{ij} a_j \right) \frac{1}{a_i}$$

Parameters

- [in] elems: discretized cavity
- [in] gf: a Green's function

Private Members

double **factor_**

Scaling factor for the diagonal elements of the matrix representation of the S operator

4.5.4 Numerical

class *pcm*::bi_operators::**Numerical** : **public** *pcm*::*IBoundaryIntegralOperator*

Implementation of the single and double layer operators matrix representation using one-point collocation.

Calculates the diagonal elements of S and D by collocation, using numerical integration.

Author Roberto Di Remigio

Date 2015, 2016

Private Functions

Eigen::MatrixXd **computeS_impl**(**const** std::vector<cavity::Element> &*elems*, **const** *IGreensFunction* &*gf*) **const override**

Computes the matrix representation of the single layer operator

Parameters

- `[in] elems`: list of finite elements of the discretized cavity
- `[in] gf`: a Green's function

Eigen::MatrixXd **computeD_impl**(**const** std::vector<cavity::Element> &*elems*, **const** *IGreensFunction* &*gf*) **const override**

Computes the matrix representation of the double layer operator

Parameters

- `[in] elems`: list of finite elements of the discretized cavity
- `[in] gf`: a Green's function

4.6 Helper classes and functions

4.6.1 Sphere

struct *pcm*::*utils*::**Sphere**

POD describing a sphere.

Author Roberto Di Remigio

Date 2011, 2016

Public Functions

void **scale**(double *scaling*)
Scale sphere to other units.

4.6.2 Atom

struct *pcm*::*utils*::**Atom**
A POD describing an atom.

Author Roberto Di Remigio

Date 2011, 2016

Public Members

double **charge**
Atomic charge

double **mass**
Atomic mass

double **radius**
Atomic radius

double **radiusScaling**
Scaling of the atomic radius

Eigen::Vector3d **position**
Position of the atom

std::string **element**
Name of the element

std::string **symbol**
Atomic symbol

4.6.3 ChargeDistribution

struct *pcm*::*utils*::**ChargeDistribution**
POD representing a classical charge distribution.

Author Roberto Di Remigio

Date 2016

Public Members

Eigen::VectorXd **monopoles**
Monopoles

Eigen::Matrix3Xd **monopolesSites**
Monopoles sites

Eigen::Matrix3Xd **dipoles**
Dipoles

Eigen::Matrix3Xd **dipolesSites**
Dipoles sites

Eigen::VectorXd **FQChi**
FQ electronegativities

Eigen::VectorXd **FQEta**
FQ hardnesses

Eigen::Matrix3Xd **FQSites**
FQ sites

4.6.4 Molecule

class *pcm*::**Molecule**
Class representing a molecule or general aggregate of atoms.

This class is based on the similar class available in the Mints library of Psi4

Author Roberto Di Remigio

Date 2014

Unnamed Group

Molecule &**operator=**(**const** *Molecule* &*other*)
Operators Assignment operator.

Public Functions

Molecule()
Default constructor Initialize a dummy molecule, e.g. as placeholder, see ICavity.cpp loadCavity method.

Molecule(int *nat*, **const** Eigen::VectorXd &*chg*, **const** Eigen::VectorXd &*masses*, **const** Eigen::Matrix3Xd &*geo*, **const** std::vector<Atom> &*at*, **const** std::vector<Sphere> &*sph*)
Constructor from full molecular data.

This initializes the molecule in C1 symmetry

Parameters

- `[in] nat`: number of atoms
- `[in] chg`: vector of atomic charges
- `[in] masses`: vector of atomic masses
- `[in] geo`: molecular geometry (format nat*3)

- `[in] at`: vector of Atom objects
- `[in] sph`: vector of Sphere objects

`Molecule`(int *nat*, **`const`** Eigen::VectorXd &*chg*, **`const`** Eigen::VectorXd &*masses*, **`const`** Eigen::Matrix3Xd &*geo*, **`const`** std::vector<Atom> &*at*, **`const`** std::vector<Sphere> &*sph*, int *nr_gen*, std::array<int, 3> *gens*)

Constructor from full molecular data, plus number of generators and generators.

This initializes the molecule in the symmetry prescribed by nr_gen and gen. See documentation of the *Symmetry* object for the conventions.

Parameters

- `[in] nat`: number of atoms
- `[in] chg`: vector of atomic charges
- `[in] masses`: vector of atomic masses
- `[in] geo`: molecular geometry (format nat*3)
- `[in] at`: vector of Atom objects
- `[in] sph`: vector of Sphere objects
- `[in] nr_gen`: number of molecular point group generators
- `[in] gen`: molecular point group generators

`Molecule`(int *nat*, **`const`** Eigen::VectorXd &*chg*, **`const`** Eigen::VectorXd &*masses*, **`const`** Eigen::Matrix3Xd &*geo*, **`const`** std::vector<Atom> &*at*, **`const`** std::vector<Sphere> &*sph*, **`const`** *Symmetry* &*pg*)

Constructor from full molecular data and point group.

This initializes the molecule in the symmetry prescribed by pg.

Parameters

- `[in] nat`: number of atoms
- `[in] chg`: vector of atomic charges
- `[in] masses`: vector of atomic masses
- `[in] geo`: molecular geometry (format nat*3)
- `[in] at`: vector of Atom objects
- `[in] sph`: vector of Sphere objects
- `[in] pg`: the molecular point group (a *Symmetry* object)

`Molecule`(**`const`** std::vector<Sphere> &*sph*)

Constructor from list of spheres.

Molecule is treated as an aggregate of spheres. We do not have information on the atomic species involved in the aggregate. Charges are set to 1.0; masses are set based on the radii; geometry is set from the list of spheres. All the atoms are dummy atoms. The point group is C1.

Warning This constructor is to be used **exclusively** when initializing the *Molecule* in EXPLICIT mode, i.e. when the user specifies explicitly spheres centers and radii.

Parameters

- `[in] sph`: list of spheres

Molecule(**const** *Molecule* &*other*)

Copy constructor.

void **translate**(**const** Eigen::Vector3d &*translationVector*)

Given a vector, carries out translation of the molecule.

Parameters

- `translationVector`: The translation vector.

void **moveToCOM**()

Performs translation to the Center of Mass Frame.

void **rotate**(**const** Eigen::Matrix3d &*rotationMatrix*)

Given a matrix, carries out rotation of the molecule.

Parameters

- `rotationMatrix`: The matrix representing the rotation.

void **moveToPAF**()

Performs rotation to the Principal Axes Frame.

Private Members

size_t **nAtoms_**

The number of atoms in the molecule.

Eigen::VectorXd **charges_**

A vector of dimension (# atoms) containing the charges.

Eigen::VectorXd **masses_**

A vector of dimension (# atoms) containing the masses.

Eigen::Matrix3Xd **geometry_**

Molecular geometry, in cartesian coordinates. The dimensions are (# atoms * 3) Units are Bohr.

std::vector<Atom> **atoms_**

A container for all the atoms composing the molecule.

std::vector<Sphere> **spheres_**

A container for the spheres composing the molecule.

rotorType **rotor_**

The molecular rotor type.

Symmetry **pointGroup_**

The molecular point group.

4.6.5 Solvent

struct *pcm*::*utils*::**Solvent**

POD describing a solvent.

A *Solvent* object contains all the solvent-related experimental data needed to set up the Green's functions and the non-electrostatic terms calculations.

Author Roberto Di Remigio

Date 2011, 2016

Public Members

std::string **name**
Solvent name

double **epsStatic**
Static permittivity, in AU

double **epsDynamic**
Optical permittivity, in AU

double **probeRadius**
Radius of the spherical probe mimicking the solvent, in Angstrom

4.6.6 Symmetry

class Symmetry
Contains very basic info about symmetry (only Abelian groups)

Just a wrapper around a vector containing the generators of the group

Author Roberto Di Remigio

Date 2014

Private Members

int **nrGenerators_** = {0}
Number of generators

std::array<int, 3> **generators_** = {0}
Generators

int **nrIrrep_** = {1}
Number of irreps

4.6.7 Mathematical utilities

namespace pcm
PCMSolver, an API for the Polarizable Continuum Model Copyright (C) 2020 Roberto Di Remigio, Luca Frediani and collaborators.

This file is part of PCMSolver.

PCMSolver is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

PCMSolver is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with PCMSolver. If not, see http://www.gnu.org/licenses/.

For information on the complete list of contributors to the PCMSolver API, see: http://pcmsolver.readthedocs.io/

namespace utils

Functions

template<size_t **nBits**>
int **parity**(std::bitset<*nBits*> *bitrep*)

Calculate the parity of the bitset as defined by: bitrep[0] XOR bitrep[1] XOR … XOR bitrep[nBits-1]

Parameters

- `[in] bitrep`: a bitset

Template Parameters

- `nBits`: lenght of the input bitset

double **parity**(unsigned int *i*)

Returns parity of input integer. The parity is defined as the result of using XOR on the bitrep of the given integer. For example: 2 -> 010 -> 0^1^0 = 1 -> -1.0 6 -> 110 -> 1^1^0 = 0 -> 1.0

Parameters

- `[in] i`: an integer, usually an index for an irrep or a symmetry operation

It can also be interpreted as the action of a given operation on the Cartesian axes: zyx Parity 0 000 E 1.0 1 001 Oyz -1.0 2 010 Oxz -1.0 3 011 C2z 1.0 4 100 Oxy -1.0 5 101 C2y 1.0 6 110 C2x 1.0 7 111 i -1.0

bool **isZero**(double *value*, double *threshold*)

Returns true if value is less or equal to threshold

Parameters

- `[in] value`: the value to be checked
- `[in] threshold`: the threshold

bool **numericalZero**(double *value*)

Returns true if value is less than 1.0e-14

Parameters

- `[in] value`: the value to be checked

template<typename **T**>
int **sign**(*T* *val*)

This function implements the signum function and returns the sign of the passed value: -1, 0 or 1

Parameters

- `[in] val`: value whose sign should be determined

Template Parameters

- `T`: of the parameter val

void **`symmetryBlocking`**(Eigen::MatrixXd *&matrix*, PCMSolverIndex *cavitySize*, PCMSolverIndex *ntsirr*, int *nr_irrep*)

void **`symmetryPacking`**(std::vector<Eigen::MatrixXd> *&blockedMatrix*, **`const`** Eigen::MatrixXd *&fullMatrix*, int *dimBlock*, int *nrBlocks*)

Parameters

- `[out] blockedMatrix`: the result of packing fullMatrix
- `[in] fullMatrix`: the matrix to be packed
- `[in] dimBlock`: the dimension of the square blocks
- `[in] nrBlocks`: the number of square blocks

template<typename **`Derived`**>
void **`hermitivitize`**(Eigen::MatrixBase<*Derived*> *&obj_*)

Given obj_ returns 0.5 * (obj_ + obj_^dagger)

Note We check if a matrix or vector was given, since in the latter case we only want the complex conjugation operation to happen.

Parameters

- `[out] obj_`: the Eigen object to be hermitivitized

Template Parameters

- `Derived`: the numeric type of obj_ elements

void **`eulerRotation`**(Eigen::Matrix3d *&R_*, **`const`** Eigen::Vector3d *&eulerAngles_*)

Build rotation matrix between two reference frames given the Euler angles.

We assume the convention $R = Z_3X_2Z_1$ for the ordering of the extrinsic elemental rotations (see http://en.wikipedia.org/wiki/Euler_angles) The Euler angles are given in the order ϕ, θ, ψ. If we write c_i, s_i $i = 1, 3$ for their cosines and sines the rotation matrix will be:

$$R = \begin{pmatrix} c_1c_3 - s_1c_2s_3 & -s_1c_3 - c_1c_2s_3 & s_2s_3 \\ c_1s_3 + s_1c_2c_3 & -s_1s_3 + c_1c_2c_3 & -s_2c_3 \\ s_1s_2 & c_1s_2 & c_2 \end{pmatrix}$$

Eigen's geometry module is used to calculate the rotation matrix

Parameters

- `[out] R_`: the rotation matrix
- `[in] eulerAngles_`: the Euler angles, in degrees, describing the rotation

double **`linearInterpolation`**(**`const`** double *point*, **`const`** std::vector<double> *&grid*, **`const`** std::vector<double> *&function*)

Return value of function defined on grid at an arbitrary point.

This function finds the nearest values for the given point and performs a linear interpolation.

Warning This function assumes that grid has already been sorted!

Parameters

- `[in] point`: where the function has to be evaluated
- `[in] grid`: holds points on grid where function is known
- `[in] function`: holds known function values

double **`splineInterpolation`**(**`const`** double *point*, **`const`** std::vector<double> *&grid*, **`const`** std::vector<double> *&function*)

Return value of function defined on grid at an arbitrary point.

This function finds the nearest values for the given point and performs a cubic spline interpolation.

Warning This function assumes that grid has already been sorted!

Parameters

- `[in] point`: where the function has to be evaluated

- `[in] grid`: holds points on grid where function is known
- `[in] function`: holds known function values

template<typename **Derived**>
void **print_eigen_matrix**(**const** Eigen::MatrixBase<*Derived*> &*matrix*, **const** std::string &*fname*)

Prints Eigen object (matrix or vector) to file.

Note This is for debugging only, the format is in fact rather ugly. Row index Column index Matrix entry 0 0 0.0000

Parameters
- `[in] matrix`: Eigen object
- `[in] fname`: name of the file

Template Parameters
- `Derived`: template parameters of the MatrixBase object

Eigen::MatrixXd **prune_zero_columns**(**const** Eigen::MatrixXd &*incoming*, **const** Eigen::Matrix<bool, 1, Eigen::Dynamic> &*filter*)

Prune zero columns from matrix.

Outgoing matrix has the same number of rows as the incoming.

Parameters
- `[in] incoming`: Matrix to be pruned
- `[in] filter`: indexing array for pruning

Eigen::VectorXd **prune_vector**(**const** Eigen::VectorXd &*incoming*, **const** Eigen::Matrix<bool, 1, Eigen::Dynamic> &*filter*)

Prune zero elements from Vector.

Parameters
- `[in] incoming`: VectorXd to be pruned
- `[in] filter`: indexing array for pruning

namespace cnpy

namespace custom

Custom overloads for cnpy load and save functions

Functions

template<typename **Scalar**, int **Rows**, int **Cols**>
void **npy_save**(**const** std::string &*fname*, **const** Eigen::Matrix<*Scalar*, *Rows*, *Cols*> &*obj*)

Save Eigen object to NumPy array file.

Parameters
- `fname`: name of the NumPy array file
- `obj`: Eigen object to be saved, either a matrix or a vector

Template Parameters
- `Scalar`: the data type of the matrix to be returned. Default is double
- `Rows`: number of rows in the Eigen object. Default is dynamic e
- `Cols`: number of columns in the Eigen object. Default is dynamic

template<typename **Scalar**, int **Rows**, int **Cols**>
void **npz_save**(**const** std::string &*fname*, **const** std::string &*name*, **const** Eigen::Matrix<*Scalar*, *Rows*, *Cols*> &*obj*, bool *overwrite* = false)

Save Eigen object to a compressed NumPy file.

Parameters

- `fname`: name of the compressed NumPy file
- `name`: tag for the given object in the compressed NumPy file
- `obj`: Eigen object to be saved, either a matrix or a vector
- `overwrite`: if file exists, overwrite. Appends by default.

Template Parameters

- `Scalar`: the data type of the matrix to be returned. Default is double
- `Rows`: number of rows in the Eigen object. Default is dynamic
- `Cols`: number of columns in the Eigen object. Default is dynamic

template<typename **`Scalar`**>
Eigen::Matrix<*Scalar*, Eigen::Dynamic, Eigen::Dynamic> **`npy_to_eigen`**(**`const`** NpyArray &*npy_array*)

Load NpyArray object into Eigen object.

Todo:

Extend to read in also data in row-major (C) storage order

Return An Eigen object (matrix or vector) with the data

Warning We check that the rank of the object read is not more than 2 Eigen cannot handle general tensors.

Parameters

- `npy_array`: the NpyArray object

Template Parameters

- `Scalar`: the data type of the matrix to be returned. Default is double

template<typename **`Scalar`**>
Eigen::Matrix<*Scalar*, Eigen::Dynamic, Eigen::Dynamic> **`npy_load`**(**`const`** std::string &*fname*)

Load NumPy array file into Eigen object.

Todo:

Extend to read in also data in row-major (C) storage order

Return An Eigen object (matrix or vector) with the data

Parameters

- `fname`: name of the NumPy array file

Template Parameters

- `Scalar`: the data type of the matrix to be returned. Default is double

# Namespaces

We use namespaces to delimit the visibility of functions and classes defined in the various subdirectories of the project. Namespaces provide a convenient layered structure to the project and we use them as a convention to signal which functions and classes are supposed to be used in any given layer. The top-level namespace is called *pcm* and includes all functions and classes that can be called from the outside world, i.e. a C++ API. Each subdirectory introduces a new namespace of the same name, nested into *pcm*. Code that can be used _outside_ of a given subdirectory is put directly in the *pcm* namespace, i.e. the outermost layer. Finally, the namespace *detail*, at the third level of nesting, is used for functions and classes that are used exclusively within the code in a given subdirectory.

CHAPTER FIVE

REFERENCES

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Ale01] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN 0-201-70431-5.

[AZB94] Norman L Allinger, Xuefeng Zhou, and John Bergsma. Molecular mechanics parameters. *Journal of Molecular Structure: THEOCHEM*, 312(1):69–83, 1 January 1994. doi:10.1016/S0166-1280(09)80008-0.

[Cli] Marshall P. Cline. C++ FAQ. URL: http://www.parashift.com/c++-faq.

[CGL98] Marshall P. Cline, Mike Girou, and Greg Lomow. *C++ FAQs*. Addison-Wesley Longman Publishing Co., Inc., 1998. ISBN 0201309831.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1994. ISBN 0-201-63361-2.

[Kop08] Joachim Kopp. Efficient numerical diagonalization of hermitian 3x3 matrices. *Int. J. Mod. Phys. C*, 19(03):523–548, 2008. doi:10.1142/S0129183108012303.

[RCC+92] A. K. Rappe, C. J. Casewit, K. S. Colwell, W. A. Goddard, and W. M. Skiff. UFF, a full periodic table force field for molecular mechanics and molecular dynamics simulations. *J. Am. Chem. Soc.*, 114(25):10024–10035, 1992. URL: http://pubs.acs.org/doi/abs/10.1021/ja00051a040, doi:10.1021/ja00051a040.

[Sut99] Herb Sutter. *Exceptional C++: 47 engineering puzzles, programming problems, and solutions*. Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0-201-61562-2.

[SA04] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series)*. Addison-Wesley Professional, 2004. ISBN 0321113586.

[TMC05] Jacopo Tomasi, Benedetta Mennucci, and Roberto Cammi. Quantum mechanical continuum solvation models. *Chem. Rev.*, 105(8):2999–3093, 2005. doi:10.1021/cr9904009.

[WAB+14] Greg Wilson, D a Aruliah, C Titus Brown, Neil P Chue Hong, Matt Davis, Richard T Guy, Steven H D Haddock, Kathryn D Huff, Ian M Mitchell, Mark D Plumbley, Ben Waugh, Ethan P White, and Paul Wilson. Best practices for scientific computing. *PLoS Biol.*, 12(1):e1001745, 2014. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3886731\T1\textbackslash{}&tool=pmcentrez\T1\textbackslash{}&rendertype=abstract, doi:10.1371/journal.pbio.1001745.

[Bondi64] A. Bondi. van der Waals Volumes and Radii. *J. Phys. Chem.*, 68(3):441–451, 1964. URL: http://pubs.acs.org/doi/pdf/10.1021/j100785a001, doi:10.1021/j100785a001.

[CancesMennucci98] Eric Cancès and Benedetta Mennucci. New Applications of Integral Equations Methods for Solvation Continuum Models: Ionic Solutions and Liquid Crystals. *J. Math. Chem.*, 23:309–326, 1998. doi:10.1023/A:1019133611148.

[MantinaChamberlinValero+09] Manjeera Mantina, Adam C. Chamberlin, Rosendo Valero, Christopher J. Cramer, and Donald G. Truhlar. Consistent van der Waals Radii for the Whole Main Group. *J. Phys. Chem. A*, 113:5806–5812, 2009. URL: http://pubs.acs.org/doi/pdf/10.1021/jp8111556, doi:10.1021/jp8111556.

INDEX