Finanziato dall'Unione europea NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani PIANO NAZIONALE DI RIPRESA E RESILIENZA
ICSC Centro Nazionale di Ricerca in HPC, Big Data and Quantum Computing

Nano-particle Transition Matrix Code User's Handbook

Code release version 0.10.5

G. La Mura, G. Mulas, S. Rezaei, R. Saija

December 2025

Contents

Scope of the document

This document is provided along with the `NP_TMcode` suite as a general guide to the software possibilities, main goals and instructions of use. Being intended as a quick guide, this document does not dig in the details of the code implementation. More detailed documentation, explaining the structure of the code and the role of its main components (algorithms, data structures, functions and variables) is given in the form of `doxygen`-handled inline documentation, README files and wiki pages, as well as in associated technical references.

This handbook is organized in chapters:

- Chapter 1 presents the application suite and a description of its functionality;

- Chapter 2 describes the input and output formats used to define the models and store the results of the calculations;

- Chapter 3 illustrates how to use the code auxiliary *Python* scripts to process existing models;

- Chapter 4 discusses the correct procedures that need to be followed to build and solve user-defined models.

The code is registered under the *Astrophysics Source Code Library* (ASCL, `https://ascl.net/`), as `https://ascl.net/2510.003`, and it is publicly available as an `OpenSource` project under a GNU General Public License (GNU GPLv3) distributed via gitLab at:

`https://www.ict.inaf.it/gitlab/giacomo.mulas/np_tmcode`

An alternative distribution, with a smaller set of test cases, can be obtained from gitHub at:

`https://github.com/ICSC-Spoke3/NP_TMcode`

Chapter 1

The NP_TMcode applications

1.1 Package description

NP_TMcode is a software package that provides a set of applications designed to solve the problem of radiation scattering and absorption by material particles, using the *transition matrix* formalism (T-matrix, Waterman, 1971; Mishchenko et al., 1996; Borghese et al., 2007). The full theoretical solution of the problem of scattering and absorption of radiation by material particles is only possible in a limited set of simplified geometries. The case of rigorous spherical symmetry can be solved in the framework of the *Mie theory* (Mie, 1908). Extensions of this analytical treatment to the cases of ellipsoidal and cylindrical symmetry are also possible. Conversely, the solution of non-trivially symmetric cases, like those applying to the vast majority of realistic situations, cannot be derived from theory and requires numerical approaches. The T-matrix formalism is among the most powerful methods, because it inherently supports all possible scattering geometries and it grants for a high level of detail in the representation of particle models, but it has demanding computational requirements compared to alternative methods such as the *Discrete-Dipole Approximation* (DDA, Purcell and Pennypacker, 1973; Draine and Flatau, 1994), the *Modified Mean-Field* theory (MMFT, Tazaki and Tanaka, 2018), and the *Finite Difference Time Domain* method (FDTD, Yee, 1996; Taflove and Hagness, 2005). NP_TMcode addresses the complexity of the T-matrix formalism by introducing a new implementation of the

method that takes advantage from parallel hardware architectures, such as GPUs and multi-core CPUs, leveraging the capabilities of modern hardware architectures to speed up the calculations and to allow for the solution of realistic particle models.

The package is composed by four core applications, named `np_sphere`, `np_cluster`, `np_inclusion`, and `np_trapping`, with the addition of a set of *Python* scripts, used to assist the user in the preparation of particle models and in the analysis of the calculation results. The role of these applications is to solve the following cases:

- `np_sphere` computes the scattering and absorption solution of a single particle made up by one or more material layers in spherical symmetry using the Mie theory,

- `np_cluster` solves the case of a generic particle represented by an arbitrary combination of non-intersecting spherical components using the T-matrix formalism,

- `np_inclusion`, formally similar to `np_cluster`, but adding a spherical outer coating around the aggregate,

- `np_trapping`, which, instead, calculates the forces and the torques exerted on a particle trapped by a focused laser beam with the "optical tweezers" technique.

For convenience, these four core applications follow the same execution scheme, which uses a set of machine readable configuration files, to describe the input model, and then writes the results in an output folder. The output consists of the integrated and differential cross-sections, the asymmetry parameters and the forces and torques exerted on a model particle exposed to a radiation field. All the information is stored in `HDF5` binary files, as well as in plain text formatted files. Optionally, the T-matrix of a model particle, which contains all the information concerning the scattering process for a single wavelength, can also be saved. A scheme of the situations addressed by the `NP_TMcode` applications is shown in Fig. 1.1.

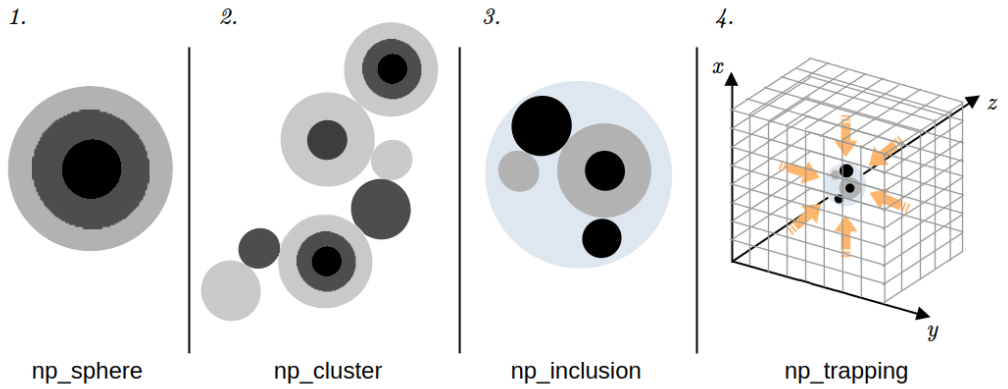

Figure 1.1: Schematic representation of the modeling possibilities of the `NP_TMcode` application suite. The case of a single particle with spherical symmetry and one or more concentric layers is addressed by `np_sphere` in the framework of the Mie theory (panel 1). `np_cluster` solves the case of aggregates of arbitrary numbers of spherical monomers, with the possibility of using monomers with different compositions, concentric layer structures and sizes (panel 2). `np_inclusion` is similar to `np_cluster`, except for embedding the aggregate in an external spherical coating (panel 3). Finally, `np_trapping` solves for the forces and torques exerted on a particle trapped by a radiation beam (panel 4).

1.2 Package structure

`NP_TMcode` is developed and maintained through the `git` versioning system. As a consequence, the package is subdivided in a folder structure that distinguishes the source code and the application data files, which are subject to revisions, from the build structure, which, instead, hosts user defined and system dependent data. The structure of the package, starting from `NP_TMcode v0.10.3`, is summarized in Table 1.1.

As shown by Table 1.1, the software package consists of five mandatory folders (`build`, `doc`, `ref_data`, `src`, and `test_data`), together with an optional folder named `containers`. The `build` folder contains the configuration script and the necessary information to build the executable binary

Table 1.1: The NP_TMcode package folder structure.

```
np_tmcode
    |          build
               |          cluster     libnptm   testing
                          inclusion    sphere   trapping
          containers(*)
               |          docker
                          singularity
              doc
               |            src
                           build
            ref_data
              src
               |          cluster     libnptm   testing
                          include      scripts  trapping
                          inclusion    sphere
            test_data
               |          cluster      sphere
                          inclusion    trapping
```

(*) The containers folder is only available via the gitLab hosted distribution.

files. Except for these pre-existing files, any other contents of this folder are ignored by the git system and are not subject to version control. The doc, ref_data, src, and test_data folders, instead, contain respectively the code inline documentation (managed via doxygen), some reference data (such as, for instance, the optical constants of various materials as a function of wavelength), the source code of the software, and some data to perform consistency tests. The package is configured in such a way that each application has its own sub-folder structure in the build, source and test data sections, with np_sphere mapping to sphere folders, np_cluster to cluster folders, np_inclusion to inclusion folders, and np_trapping to trapping folders. The libnptm folders refer to the code proprietary libraries (both in source and binary versions), while the include folder contains the code

header files. The `src/scripts` folder stores procedural scripts that, being written in interpreted languages such as *Python*, do not need to be compiled before execution.

The `containers` folder is a special section, storing pre-built binary libraries and definition files from which versions of `NP_TMcode` designed to work with the `docker` and `singularity` container managers can be accessed. These versions can then be used to run the code on different system architectures, not limited to Linux operating systems, although introducing the additional costs and limitations of the container interface towards the host machine. Due to the large size of this special folder, it is only distributed through the `gitLab` development repository, while the other code distribution channels offer a lighter package without this feature.

All the main sections of the package are further documented in local `README.md` files, written in `MarkDown` complying plain text syntax, while the inline documentation presents detailed explanations of the code structure, as extracted from properly formatted comment sections in the source code.

1.3 `NP_TMcode` installation

1.3.1 Prerequisites

The native installation of `NP_TMcode` is based on the typical configuration and building stages of UNIX software packages. In order to correctly build the software package, the host system must provide a `bash`-compliant shell, a `make` command (the recommended option is `GNU-make`), a *C++* compiler and, optionally, a *FORTRAN* compiler (recommended ones are `g++` and `gfortran` with version ≥ 13.0, although some degree of compatibility is offered also for `clang` and *Intel* compilers, as well). `NP_TMcode` needs a working installation of the `HDF5` libraries to handle its binary output files, while optional dependencies on various implementations of `LAPACK` (Anderson et al., 1999), `MAGMAv2` (Bosma et al., 1997), `OpenMP` (Dagum and Menon, 1998), and `MPI` (The MPI Forum, 1993) can be added to achieve increasing levels of code optimization and parallel execution capabilities.

1.3.2 Obtaining the code

NP_TMcode is publicly available through the *Astrophysics Source Code Library* at https://ascl.net/2510.003, which refers to the repository websites of gitHub and gitLab:

- https://www.ict.inaf.it/gitlab/giacomo.mulas/np_tmcode (development version with full container images)
- https://github.com/ICSC-Spoke3/NP_TMcode (light-weight distribution version, without containers)

Both websites offer the opportunity to access release bundles, to download compressed archives of the source code, or to obtain maintainable versions via the git system. The recommended way to obtain the code is to download the most recent release published on the gitHub portal.

More up to date maintenance versions of the code can be obtained by using the git clone commands with the repository addresses provided by the gitHub and the gitLab project homepage. Code downloaded via the git clone command will be connected with the source repository where it was originally taken from. This allows for keeping the downloaded version up to date with respect to the source repository by invoking git pull commands each time a new feature has been added to the main code branch.

Another optional way to download the code is to get a compressed archive of the source code from the repository's browser interface. This method creates a local version of the code which cannot be synchronized via git pull commands. However, this method is not recommended, due to the experimental nature of the features that can be included in the code between different releases. Downloading source code distributions from branches that are not flagged as the main ones is not recommended, except for users who plan to contribute to the development.

1.3.3 Configuring and building the code

Configuration and building stages need to be executed from the package's build folder on a machine that can access the same software resources as

the ones that are actually intended to run the code. While this is trivially granted for most user workstations, it might not be the case for computing farms, where the login nodes may have substantially different configurations with respect to the computing ones.

The configuration stage is handled by a shell script named `configure`,[1] whose role is to scan the system and to detect the most convenient compiler flags to effectively build the code. In general, the package basic configuration and building stage should work simply by issuing:

```
build$ ./configure
build$ make -j
```

The script will then try to detect the most convenient configuration and build `NP_TMcode` accordingly. In some cases, the configuration script may not be able to automatically identify the best possible configuration (e.g., if the host system does not provide the `pkgconfig` manager or if the most optimized libraries reside in user specific location). In these cases the configuration can be improved with by providing some command line options. A complete list of the available options can be seen by issuing:

```
build$ ./configure --help
```

The inline documentation can optionally be built with the command:

```
build$ make docs
```

This will populate the `np_tmcode/doc/build/html` folder with the inline documentation of the code, in the form of a browser website rooted in the file `index.html`. The source code to create a `PDF` version of the documentation, through the LaTeX system will also be placed in the `np_tmcode/doc/build/latex` folder. If the host system supports LaTeX, this step can be completed by going to the `np_tmcode/doc/build/latex` folder and further issuing

```
build$ make
```

[1] `configure` is actually a symbolic link to the real configuration script, which is `configure.sh`.

1.4 Testing the installation

Testing NP_TMcode installation requires access to the same hardware and software resources for which the code has been built. If the code is executed on a local workstation, this condition is already met. If, instead, the code is installed on a computing farm, the test should be run on a computing node providing all the features selected during the configuration stage (such as, for instance, access to GPUs). A minimal test can then be run as follows:

```
build$ cd cluster
cluster$ [mpirun -n 1] ./np_cluster
... OMITTED OUTPUT LOG ...
cluster$ ../../src/scripts/pycompare.py --cfile c_OCLU --ffile \
         ../../test_data/cluster/OCLU
```

where the part in square brackets ("`mpirun -n 1`") is only required if the code was configured and built to run with *MPI*.[2] If everything was correctly set up, the result of the test should close with the following message:

```
SUCCESS: c_OCLU is consistent with ../../test_data/cluster/OCLU
```

otherwise, a message like:

```
FAILURE: c_OCLU is not consistent with ../../test_data/cluster/OCLU
```

will appear. In this case, the installation process probably had issues that need to be further investigated.

The details of the command executions and of the related command line arguments will be clarified in the following chapters, where the workflow of the code is further explained.

[2] Actually, this is only necessary for some implementations of *MPI*, since, in case the "`mpirun -n 1`" command is omitted, the system implicitly assumes that the application is run as a single process.

Chapter 2

Model definition and results

2.1 General I/O policy

Due to consistency testing requirements, `NP_TMcode` handles its input and output definitions in two fundamental ways: a legacy format, based on strictly formatted ASCII files, which can be directly used also by the original *FORTRAN 66* applications (included in the package), and a new system based on `YAML` compliant model configuration files and `HDF5` binary output files. The persistence of the legacy I/O approach is justified by the need to verify that the results produced by the new `NP_TMcode` implementation of the T-matrix formalism is consistent with the original one during the development phase.

All `NP_TMcode` applications adopt the same fundamental execution syntax, which is:

```
DATA_DIR$ INSTALL_DIR/NP_APP CONFIG_1 CONFIG_2 OUTPUT_DIR
```

where the capitalized elements have the following meanings:

- `DATA_DIR`: A working directory of the file-system, which serves as the root path where the code is assumed to be running and should contain the configuration files, or provide reading access to their location.

- `INSTALL_DIR`: The location on the file-system where the `NP_TMcode` application binaries were built.

- `NP_APP`: The application that is going to be executed (one of `np_sphere`, `np_cluster`, `np_inclusion`, or `np_trapping`).

- `CONFIG_1`: First code configuration file, generally containing the calculation description.

- `CONFIG_2`: Second code configuration file, typically used to describe the particle model and to add runtime options.

- `OUTPUT_DIR`: A pre-existing folder pointing to the location of the file-system where the results will be written.

`NP_TMcode` applications can use the current working directory as output folder, but with the caveat that the names of the output files are still hard-coded in the application binaries. As a consequence, multiple calculations executed on the same output folder will lead to a **silent overwriting of the previous results**, unless they were manually renamed before the new calculation is started.

The presence of runtime settings in the second configuration file is optional and it is only managed by the *C++* implementation of `NP_TMcode`. The legacy *FORTRAN 66* applications are not affected by runtime settings and simply ignore them, since they do not even parse the file lines following the block of their original input parameters.

Invoking the code with a different argument pattern **will result in the applications running in test mode,** which may either fail or produce wrong results. Also using a non-existing `OUTPUT_DIR`, or an output directory that does not grant writing permissions, will lead to runtime failures. In `NP_TMcode v0.10.5` it is up to the user to make sure that all the requirements are met, although more advanced exception handling procedures may be adopted in later versions.

2.2 The reference data files

In order to obtain a proper solution of the scattering and absorption properties of matter particles, the code needs a definition of the optical properties of the involved materials. These can be defined in several ways, but the most

convenient one is to provide tabulated values of the material's complex dielectric constant as a function of incident radiation wavelength. For the moment being, these tables can be provided in the form of plain text comma separated value files (CSV). A few examples of such reference data are available in the np_tmcode/ref_data folder. The correct format includes three columns, corresponding, respectively, to the radiation wavelength in meters,[1], to the real part of the dielectric constant, and to its imaginary part. Initial lines starting with a *hash* character ('#') are considered header lines, while the data lines must comply with the format:

$$\lambda,\mathfrak{Re}(\varepsilon),\mathfrak{Im}(\varepsilon)$$

where real numbers can be specified either as floating point values or as scientific notation numbers, with an exponent factor identified by 'e' or 'E' characters.

NP_TMcode can access optical property table files anywhere in the filesystem (provided that reading access to the pointed location is granted), so there is no need to place additional data files in the ref_data folder, before using them. The files distributed alongside the software only serve as examples on how this type of information should be defined. Once the optical properties of the materials are defined, the code is able to perform calculations on the whole wavelength range covered by the reference grid, either using the grid's tabulated values, or by performing interpolations, if necessary.

2.3 The YAML configuration files

As outlined in Section 2.1, each NP_TMcode application needs two configuration files, typically describing the calculation settings and the particle model. These are plain text files, read by the legacy code implementation, that can be modified with any text editor. However, the strict requirements of the legacy input parsing system and the complexity of the data that describe advanced particle models make the direct use of these configuration files a

[1] NP_TMcode applications use the units of measurement defined by the *International System* for their input and output.

bit impractical, even if possible. In order to improve the clarity of the input definition, `NP_TMcode` can use `YAML` syntax to define the necessary information. The result is a machine readable file format, with sections, comments and tags that clarify the meaning of all the parameters, thus assisting users in the definition of their own models.

The `YAML` format was chosen because it supports arbitrarily complex, nested structures, using labels to tag sections and allowing for comments. The standard way to mark a comment in `YAML` is to prepend it with the hash character ('`#`'). The nesting of structures, instead, is controlled by the level of indentation. Model configuration files can be then converted into native legacy code configuration files, using the `model_maker.py` script located in the `src/scripts` folder of the package. An example configuration file, with the meaning of all the contents explained by means of comment lines, is reported below for the case of an aggregate of four spheres with predetermined types and positions:

```
system_settings:
  max_host_ram : 0 # host RAM in GB (0 disables limits)
  max_gpu_ram  : 0 # GPU RAM in GB (0 disables limits)

input_settings:
  input_folder : "." # Folder to write input configuration
  spheres_file : "DEDFB" # Name for configuration file 1
  geometry_file: "DCLU" # Name for configuration file 2

output_settings:
  jwtm         : 1 # Index of the wavelength to save T-matrix
                   # (1 is first, 0 disables saving T-matrix)

particle_settings:
  application : "CLUSTER" # One of CLUSTER, SPHERE, INCLUSION
  n_spheres   : 4 # Number of spheres in the aggregate
  n_types     : 4 # Number of different sphere types
  sph_types   : [ 1, 2, 3, 4 ] # Which type is each sphere
  n_layers    : [ 1, 1, 1, 1 ] # Number of layers in each type
  # Spherical monomer radii in m (one size for each type)
  radii       : [ 5.0e-8, 4.0e-8, 7.5e-8, 3.0e-8 ]
  # Layer fractional radii (one per layer in each type)
  rad_frac    : [ [ 1.0 ], [ 1.0 ], [ 1.0 ], [ 1.0 ] ]
```

```
  # Index of the dielectric functions (one per layer in each
  # type)
  # 1 is first file in ‘dielec_file‘, 2 is second ...
  dielec_id   : [ [ 1 ], [ 1 ], [ 1 ], [ 1 ] ]

material_settings:
  diel_flag   : -1 # Flag to define what type of dielectric
                   # properties are in use (-1 is derived from
                   # reference scale, while 0 uses tabulated data)
  extern_diel : 2.3104e0 # External medium dielectric constant
  dielec_path : "." # Dielectric functions files folder
  # List of dielectric functions files (used if diel_flag = 0)
  dielec_file : [ ]
  # Dielectric function files format (same for all files)
  dielec_fmt  : [ "CSV" ]
  # Matching method between optical functions and radiation
  # wavelengths
  # INTERPOLATE: the functions are interpolated on
  #              wavelengths
  # GRID: only the wavelengths with defined functions are
  #       computed
  match_mode  : "GRID"
  # Reference dielectric functions (used if diel_flag = -1)
  # One real and one imaginary part for each layer in each
  # type.
  diel_func   : [
    [ 3.25e00, 0.00e00 ],
    [ 3.25e00, 0.00e00 ],
    [ 3.25e00, 0.00e00 ],
    [ 3.25e00, 0.00e00 ]
    ]

radiation_settings:
  # Radiation field polarization (LINEAR | CIRCULAR)
  polarization: "LINEAR"
  scale_start : 1.0e00 # First scale to be used
  scale_end   : 2.0e00 # Last scale to be used
  # Calculation step (overridden if ‘match_mode‘ is GRID)
  scale_step  : 1.0e00
  wp          : 1.372e15 # Reference angular frequency
  xip         : 1.0e00 # Reference scale
```

```
  # Define scale explicitly (0) or in equal steps (1)
  step_flag   : 0
  scale_name  : "XI" # Type of scaling variable

geometry_settings:
  li          : 8 # Maximum internal field expansion
  le          : 8 # Maximum external field expansion
  # Number of transition layer integration points
  npnt        : 149
  # Number of non transition layer integration points
  npntts      : 300
  iavm        : 0 # Averaging mode
  isam        : 0 # Meridional plane flag
  in_th_start : 79.0 # Starting incidence azimuth angle
  in_th_step  : 10.0 # Incidence azimuth angle incr. step
  in_th_end   : 89.0 # Ending incidence azimuth angle
  in_ph_start : 0.0 # Starting incidence elevation angle
  in_ph_step  : 10.0 # Incidence elevation angle incr. step
  in_ph_end   : 10.0 # Ending incidence elevation angle
  sc_th_start : 34.0 # Starting scattered azimuth angle
  sc_th_step  : 15.0 # Scattered azimuth angle incr. step
  sc_th_end   : 49.0 # Ending scattered azimuth angle
  sc_ph_start : 5.0 # Starting scattered elevation angle
  sc_ph_step  : 5.0 # Scattered elevation angle incr. step
  sc_ph_end   : 10.0 # Ending scattered elevation angle
  x_coords    : [
    0.00e00,
    0.00e00,
    1.18882e-07,
    -5.656855e-08
    ] # Vector of sphere X coordinates in m
  y_coords    : [
    3.00e-08,
    3.00e-08,
    3.00e-08,
    -2.656855e-08
    ] # Vector of sphere Y coordinates in m
  z_coords    : [
    0.00e00,
    9.00e-08,
    3.8627e-08,
```

```
    0.00e00
    ] # Vector of sphere Z coordinates in m
```

The format of the YAML configuration files may vary slightly, depending on the application that is being configured and to some special requirements which arise, for instance, if the user requests the construction of a particle model composed by a large number of monomers with random types and positions, or in the case of custom runtime options. More details on these possibilities are presented in Chapter 4.

Once a proper configuration file has been prepared, its conversion to the actual legacy NP_TMcode application input files can be obtained by issuing:

```
DATA_DIR$ INSTALL_DIR/src/scripts/model_maker.py YAML_CONFIG
```

where INSTALL_DIR is the location where the NP_TMcode package was installed on the system.

2.4 np_sphere

np_sphere solves the scattering and absorption problem for a single particle with spherical symmetry, using the *Mie theory* approach. Its execution requires three arguments:

1. the calculation description file (conventionally named DEDFB;
2. the runtime options file (here named DSPH);
3. an existing output folder.

Any changes to the above execution scheme result in the application silently trying to run in test mode, with default arguments, and failing if the expected test files are not found in the default relative paths. In most cases, the model_maker.py script will take care to produce properly formatted configuration files, but in some situations, especially for testing or advanced script automation, it is useful to understand the structure of the actual legacy code input. np_sphere has the simplest input format among NP_TMcode codes, therefore understanding its layout is a first critical step.

2.4.1 Input files

As previously stated, there are two configuration files that control the calculation for a spherical particle. The structure of the `DEDFB` file has the following format (text marked with hash characters '#' is for commenting purposes and **must not appear in the input files**):

```
1   1  0
2 EXDC  WP  XIP  IDFC  NXI  INSTPC  XITYP
3 XI_1
4 ... # repeats for all wavelength scales
5 XI_NXI
6    1
7  NLYRS   RSPH
8 RLYR_1
9 ... # repeats for all layers in the sphere
10 RLYR_NLYRS
11 (REAL_EPS_LYR_1_XI_1,IMAG_EPS_LYR_1_XI_1)
12 (REAL_EPS_LYR_3_XI_1,IMAG_EPS_LYR_3_XI_1)
13 ... ... # repeats for all odd layers in the sphere
14 (REAL_EPS_LYR_NLYRS_XI_1,IMAG_EPS_LYR_NLYRS_XI_1)
15 ... # repeats for all wavelength scales
16 (REAL_EPS_LYR_1_XI_NXI,IMAG_EPS_LYR_1_XI_NXI)
17 (REAL_EPS_LYR_3_XI_NXI,IMAG_EPS_LYR_3_XI_NXI)
18 ... ... # repeats for all odd layers in the sphere
19 (REAL_EPS_LYR_NLYRS_XI_NXI,IMAG_EPS_LYR_NLYRS_XI_NXI)
20 0
```

The first line (labeled 1 here) contains two integer values: the number of spheres and a flag to enable an external spherical coating. For `np_sphere` these values **must always be set to** '1' (forcing one sphere) and '0' (external coating disabled), respectively. The second line contains the following information:

- `EXDC`: the dielectric constant of the external medium at the reference frequency (float).

- `WP`: the radiation field reference angular frequency (float).

- `XIP`: the radiation field reference scale (float)

- `IDFC`: the type of dielectric functions in use (integer, < 0 to scale from `XIP value`, $= 0$ to use tabulated values, > 0 for constants functions).

- `NXI`: the number of radiation field wavelengths to be evaluated (integer).

- `INSTPC`: flag for the wavelength scale definition (integer, '0' for tabulated wavelengths, '1' for equally spaced scale steps).

- `XITYP`: flag for dielectric funtion definition (integer, '1' to depend on size parameters, '2' to depend on wave numbers, '3' to depend on wavelength, '4' to depend on angular frequency, '5' to depend on radiation field energy in eV).

Lines from 3 to 5 are used to define the wavelength range of the calculation and they are subject to variations depending on the value of the `INSTPC` flag. If `INSTPC` is set to '1', line 3 must contain two floating point numbers: the first scale to be used and the scale incremental step. In this case, the radiation field will be evaluated for a total of `NXI` different wavelengths, starting from the one identified by the first scale `XI_1` and using a regular step. If, on the contrary, `INSTPC` is set to '0', the following `NXI` lines will be read to define a tabulated wavelength grid (an useful option if the goal of the calculation is to explore a spectral range with arbitrarily changing resolution). In this latter case, there is no restriction on the sorting of the explicit wavelength scales, though it is conventional to list them in some regular ascending or descending order.

After defining the radiation field, the section starting from line 6 concerns the description of monomer types. Since `np_sphere` deals with the single sphere case, only 1 type is allowed and line 6 must correspond to a single '1' integer value. Line 7, on the other hand, describes the monomer in terms of its number of layers (`NLYRS`, an integer value) and of its total radius in meters (`RSPH`, a floating point number). The following lines, from 8 to 10 must match `NLYRS` in number and contain a single floating point number, representing the fraction of radius which is reached up by the external shell of the current layer. The list of layer radii **must be ordered** starting from the innermost layer and proceeding outwards to the outermost.

The section of configuration ranging from line 11 to line 19 contains the dielectric functions of the materials composing the spherical layers. In order to properly sort the data, it is important to take into account that only odd-numbered layers are considered as composed by the declared material, while the intervening even-numbered ones are treated as *transition* layers, where the composition gradually changes from that of the inner layer to that of the outer one. Thus, if the user is interested in modeling a spherical particle composed by 2 different materials, it will be necessary to define 3 layers: one for the innermost material, one acting as transition layer, and, finally, one as the external layer. The thickness of the second layer will control the sharpness of the transition. The layer dielectric functions will be declared as pairs of real and imaginary values, listed for every odd numbered layer (following the scheme outlined by lines 11-14). Then the whole sequence repeats for every wavelength scale (unless `IDFC` < 0, in which case only the reference scale values for each layer must be given). The optical functions are matched with the wavelength scale *by position*. Therefore, the blocks of optical functions depending on wavelength **must be sorted in the same way** as the wavelength scales were provided before. The '0' character appearing in line 20 is a standard end-of-file code.

The `DSPH` file is used to define the calculation geometry and the runtime options. Its general structure has the format:

```
1    NSPH    LMAX    INPOL   NPNT   NPNTTS    ISAM
2 IN_TH_FI  IN_TH_ST IN_TH_LA SC_TH_FI  SC_TH_ST SC_TH_LA
3 IN_PH_FI  IN_PH_ST IN_PH_LA SC_PH_FI  SC_PH_ST SC_PH_LA
4  JWTM
5 0
6 # [RUNTIME OPTIONS]
7 ...
```

Here line 1 contains the following 6 elements:

- `NSPH`: the number of spheres (integer, must be 1 for `np_sphere`).

- `LMAX`: the maximum multipolar field expansion order (integer).

- `INPOL`: the incident radiation field polarization state (integer, '0' for linear, '1' for circular).

- NPNT: number of integration points for non-transition layers (integer).

- NPNTTS: number of integration points for transition layers (integer).

- ISAM: flag for geometry reference plane (integer, '0' for scattering plane, '1' for scattering plane across z-axis, > 1 for scattering plane across z-axis and fixed scattering angle).

Lines 2 and 3 describe the geometry of any differential calculations that should be carried out for specific scattering angles. Line 2 defines, respectively, the first incident radiation azimuth angle, the incident radiation azimuth angle incremental step, the last incident radiation azimuth angle, the first scattered radiation azimuth angle, the scattered radiation azimuth angle incremental step, and the last scattered radiation azimuth angle. Likewise, line 3 provides the elevation angles of the incident and of the scattered radiation, in the same order.

The last lines of the DSPH file contain an integer number, identifying the index of the wavelength for which the computed T-matrix should be saved (JWTM in line 4, with JWTM = 1 being the first wavelength and JWTM = 0 disabling the T-matrix output) and an end-of-data code ('0' in line 5). Optional runtime parameters can be added after this line (more details in Chapter 4).

2.4.2 Output files

Upon correct execution, np_sphere produces a set of output files, with names preceded by a 'c_' prefix, in order to distinguish the files written by the *C++* implementation from the equivalent ones created by the *FORTRAN 66* legacy applications. The output files are:

- c_OEDFB: a plain text file containing a log of the configuration.

- c_OSPH: the main result file in plain text format.

- c_OSPH.hd5: the same results, but stored in HDF5 binary format.

- c_TEDF: a proprietary binary file storing configuration information.

- `c_TEDF.hd5`: the same configuration information stored in `HDF5` format.

- `c_TPPOAN`: a proprietary binary file with the main results (deprecated).

In case the code was configured to use `JWTM` other than '0', the T-matrix corresponding to the wavelength identified by the `JWTM` parameter is additionally saved to a binary `HDF5` file named `c_TTMS.hdf5` and in a proprietary binary file named `c_TTMS`.[2] All the output files are written under the folder pointed to by the `OUTPUT_DIR` command line argument defined in Section 2.1. The results of any previous calculations in the output folder **are directly overwritten**, if they were not renamed to something different than the standard output file names.

Referring to the plain text version, the main output file is structured according to the following scheme (text marked with the hash character '`#`' is for commenting purposes and does not appear in the actual files, while backslash characters '\' imply line continuation):

```
## Preamble: replication of input configuration ##
READ(IR,*)NSPH,LM,INPOL,NPNT,NPNTTS,ISAM
 nsph   lm   inpol   npnt   npntts   isam
READ(IR,*)TH,THSTP,THLST,THS,THSSTP,THSLST
 th  thstp  thlst  ths thsstp  thslst
READ(IR,*)PH,PHSTP,PHLST,PHS,PHSSTP,PHSLST
 ph  phstp  phlst  phs  phsstp  phslst
READ(IR,*)JWTM
 jwtm
 READ(ITIN)NSPHT
 READ(ITIN)(IOG(I),I=1,NSPH)
 READ(ITIN)EXDC,WP,XIP,IDFC,NXI
 READ(ITIN)(XIV(I),I=1,NXI)
 READ(ITIN)NSHL(I),ROS(I)
 READ(ITIN)(RCF(I,NS),NS=1,NSH)

 REFR. INDEX OF EXTERNAL MEDIUM=  exdc^2
  LIN|CIRC # depending on INPOL setting

# The following line only appears if XITYP = 1
```

[2] As for the case of `c_TPPOAN`, the use of proprietary binary format is deprecated and will disabled in the future.

```
 VK=ref_vacuum_k, XI IS SCALE FACTOR FOR LENGTHS

# The following section repeats for all wavelength scales
========== JXI =  1 ====================
# If XITYP = 1:
  XI=xi_value
# otherwise:
  VK=2*pi/lambda, XI=ang_freq/ref_ang_freq
     SPHERE  1 # ONLY 1 SPHERE IN THIS CASE
# If the sphere has only one layer
  SIZE=2*pi/lambda*rsph, REFRACTIVE INDEX=Real(n) Imag(n)
# otherwise:
  SIZE=2*pi/lambda*rsph
# Integrated cross-sections and albedoes
 ----- SCS ----- ABS ----- EXS ----- ALBEDS --
 scattering_cs absorption_cs extinction_cs albedo
# Ratios of cross-sections to geometrical sections
 ---- SCS/GS -- ABS/GS -- EXS/GS ---
 scattering_cs/gs absorption_cs/gs extinction_cs/gs
 FSAS = Re(FSAS)  Im(FSAS) # forward scattering amplitude
 QSCHU=4*pi*Imag(FSAS)/GS, PSCHU=4*pi*Real(FSAS)/GS, SOMAG= \
   Abs(FSAS)/(4*pi*k^3) # k is the wave vector
# Asymmetry parameter and radiation pressure cross-section
  COSAV=asymmetry_parameter, RAPRS=radiation_pressure_cs
# Cross-section contributions to extinction and scattering torque
# components along radiation propagation vector for polarization
# state 2 (parallel to polarization plane in linear polarization
# or co-rotating with polarization vector for circular polarization.
  IPO= 1, TQEk=ext_torque_k_cs, TQSk=scat_torque_k_cs
# Cross-section contributions to extinction and scattering torque
# components along radiation propagation vector for polarization
# state 2 (perpendicular to polarization plane in linear
# polarization or counter-rotating against polarization vector
# for circular polarization.
  IPO= 2, TQEk=ext_torque_k_cs, TQSk=scat_torque_k_cs
# Differential values (repeating for each scattering angle)
********* JTH =  1, JPH =  1, JTHS =  1, JPHS =  1 ***********
# Incidence and scattering directions
  TIDG=inc_theta, PIDG=inc_phi, TSDG=scat_theta, PSDG=scat_phi
  SCAND=scattering_angle # all angles in degrees
# Control parameters on incidence and scattering directions
```

```
# referred to meridional plane (cfr. Borghese et al. 2007,
# Sec. 2.7)
  CFMP=inc_cpar_mp, SFMP=sca_cpar_mp
# Control parameters on incidence and scattering directions
# referred to scattering plane (cfr. Borghese et al. 2007,
# Sec. 2.7)
  CFSP=inc_cpar_sp, SFSP=sca_cpar_sp
# x,y,z components of unitary vector perpendicular to
# incidence plane
  UNI=(uni_x, uni_y, uni_z)
# x,y,z components of unitary vector perpendicular to
# scattering plane
  UNS=(uns_x, uns_y, uns_z)
     SPHERE  1
# Complex differential scattering amplitude of the sphere
# along scattering direction: sas11 has the same polarization
# of the parallel incident field; sas21 has perpendicular
# polarization to the parallel incident field
  SAS(1,1)= Re(sas11) Im(sas11), SAS(2,1)= Re(sas21) Im(sas21)
# sas12 has perpendicular polarization to the perpendicular
# incident field; sas22 has the same polarization of the
# perpendicular incident field
  SAS(1,2)= Re(sas12) Im(sas12), SAS(2,2)= Re(sas22) Im(sas22)
# Cross-section contributions to the x,y,z components of the
# radiation forces exerted on sphere
  Fx=rad_force_x_cs, Fy=rad_force_y_cs, Fz=rad_force_z_cs
# Mueller transformation matrix for connection of Stokes
# parameters from incident field to scattered one referred to
# meridional plane (phase matrix, cfr. Borghese et al. 2007,
# Sec. 2.8.1)
  MULS
         MULS11    MULS12    MULS13    MULS14
         MULS21    MULS22    MULS23    MULS24
         MULS31    MULS32    MULS33    MULS34
         MULS41    MULS42    MULS43    MULS44
# Mueller transformation matrix for connection of Stokes
# parameters from incident field to scattered one referred to
# scattering plane (scattering matrix, cfr. Borghese et al.
# 2007, Sec. 2.8.1)
  MULSLR
         MULSLR11  MULSLR12  MULSLR13  MULSLR14
```

```
        MULSLR21  MULSLR22  MULSLR23  MULSLR24
        MULSLR31  MULSLR32  MULSLR33  MULSLR34
        MULSLR41  MULSLR42  MULSLR43  MULSLR44
... ... # differential values for all other scattering angles
... # other JXI blocks up to a total of NXI scales
```

The HDF5 binary version of the output contains the same information as the plain text output, but in the form of vectors of equivalent values, arranged in order of increasing wavelength and, if necessary, with the same sequence of computed directions. The only difference is that the HDF5 binary file does not contain repeating labels and stores numeric values using integer and double precision floating point types, depending on the nature of the values being stored, thus resulting in more efficient use of disk space.

2.5 np_cluster

The input and output filef for `np_cluster` are similar to those used by `np_sphere`, but with the addition of the necessary extensions to describe an item that is composed by more than one constituent.

2.5.1 Input files

The calculation configuration file is still conventionally named DEDFB, but its structure, for an aggregate of spheres, must be:

```
 NSPH  0
EXDC  WP  XIP  IDFC  NXI  INSTPC  XITYP
XI_1
... # repeating for all wavelengths
XI_NXI
  ID_1 ID_2 ... ID_NSPH
 NLYRS_ID_1   RSPH_ID_1
RLYR_1
... ... # repeating for all layers in sphere type 1
RLYR_NLYRS_ID_1
 NLYRS_ID_2   RSPH_ID_2
RLYR_1
... ... # repeating for all layers in sphere type 2
RLYR_NLYRS_ID_2
```

```
15 ... # repeating for all sphere types
16  NLYRS_ID_NSPH   RSPH_ID_NSPH
17 RLYR_1
18 ... ... # repeating for all layers in sphere type ID_NSPH
19 RLYR_NLYRS_ID_NSPH
20 (REAL_EPS_ID_1_LYR_1_XI_1,IMAG_EPS_ID_1_LYR_1_XI_1)
21 (REAL_EPS_ID_1_LYR_3_XI_1,IMAG_EPS_ID_1_LYR_3_XI_1)
22 ... ... ... # repeating for all the odd layers of sphere type 1
23 (REAL_EPS_ID_1_LYR_NLYRS_XI_1,IMAG_EPS_ID_1_LYR_NLYRS_XI_1)
24 ... ... # repeating for all sphere types
25 (REAL_EPS_ID_NSPH_LYR_1_XI_1,IMAG_EPS_ID_NSPH_LYR_1_XI_1)
26 (REAL_EPS_ID_NSPH_LYR_3_XI_1,IMAG_EPS_ID_NSPH_LYR_3_XI_1)
27 ... ... ... # repeating for all the odd layers of sphere type ID_NSPH
28 (REAL_EPS_ID_NSPH_LYR_NLYRS_XI_1,IMAG_EPS_ID_NSPH_LYR_NLYRS_XI_1)
29 ... # repeating for all wavelengths
30 (REAL_EPS_ID_1_LYR_1_XI_NXI,IMAG_EPS_ID_1_LYR_1_XI_NXI)
31 (REAL_EPS_ID_1_LYR_3_XI_NXI,IMAG_EPS_ID_1_LYR_3_XI_NXI)
32 ... ... ... # repeating for all the odd layers of sphere type 1
33 (REAL_EPS_ID_1_LYR_NLYRS_XI_NXI,IMAG_EPS_ID_1_LYR_NLYRS_XI_NXI)
34 ... ... # repeating for all sphere types
35 (REAL_EPS_ID_NSPH_LYR_1_XI_NXI,IMAG_EPS_ID_NSPH_LYR_1_XI_NXI)
36 (REAL_EPS_ID_NSPH_LYR_3_XI_NXI,IMAG_EPS_ID_NSPH_LYR_3_XI_NXI)
37 ... ... ... # repeating for all the odd layers of sphere type ID_NSPH
38 (REAL_EPS_ID_NSPH_LYR_NLYRS_XI_NXI,IMAG_EPS_ID_NSPH_LYR_NLYRS_XI_NXI)
39 0
```

The meaning of lines from 1 to 5 is exactly the same as in the case of `np_sphere` outlined in Sec. 2.4.1, with identical parameters. From line 6 onwards, the necessity to describe different types of spheres implies the requirement of additional information. Line 6 contains a vector of `NSPH` integer numbers, which correspond to the type that each sphere of the aggregate will belong to.[3] The number of types can be lower than `NSPH`, because a model can use multiple identical spheres. However, in order to be properly handled, the type identifiers **must be listed in ascending order** and the type identifier of the first sphere in a group **needs to be larger or equal with**

[3]While there is no hard limit in the number of spheres that can be used, the *FORTRAN 66* implementation supports up to 16 sphere IDs per line. Therefore, if `NSPH` > 16 and compatibility with legacy code is desired, line 6 should be broken in chunks of up to 16 IDs per line.

respect to the sorting index that the sphere has been assigned within the aggregate. As an example, if the user wants to model an aggregate of 4 spheres with 2 types, then:

```
   1    2    2    2
```

would mark the first sphere as type 1 and the remaining three as type 2, while[4]

```
   1    1    1    4
```

would be needed to assign the first three spheres to type 1 and the fourth to a second type.[5]

The code section between lines 7 and 19

```
 NLYRS_ID_1   RSPH_ID_1
RLYR_1
... ... # repeating for all layers in sphere type 1
RLYR_NLYRS_ID_1
 NLYRS_ID_2   RSPH_ID_2
RLYR_1
... ... # repeating for all layers in sphere type 2
RLYR_NLYRS_ID_2
... # repeating for all sphere types
 NLYRS_ID_NSPH   RSPH_ID_NSPH
RLYR_1
... ... # repeating for all layers in sphere type ID_NSPH
RLYR_NLYRS_ID_NSPH
```

contains the layer description and it must be repeated for all declared sphere types, iterating over all the layers defined in each one of them. The last section of the `DEDFB` file, namely the lines ranging from 20 to 38 contain the dielectric functions of the aggregate components. These must be listed in order, from the innermost to the outermost odd layer of every type of sphere (i.e. all the odd layers of sphere type 1 for the first wavelength, followed by all the odd layers of sphere type 2 for the first wavelength and so on).

[4]Note that, while the *C++* implementation parses vector of sphere groups IDs as a sequence of integer values, not depending on the spacing of the values, the *FORTRAN 66* implementation **needs** such values to fit in slots of 5 digits.

[5]This restriction is automatically accounted for by the `model_maker.py` script, so it must be handled only when directly editing a `DEDFB` file.

If IDFC < 0, only the reference wavelength must be declared. Otherwise, the tabulated dielectric functions must be provided for all the computed wavelengths, replicating the same sequence used for the first wavelength for a total of NXI times. The '0' character in line 39 is the usual end-of-file code.

In addition to the calculation description file, a second file, conventionally named DCLU, is required to describe the geometry of the model and to define any possible runtime option. The format of a DCLU file has the structure:

```
1  NSPH   LI   LE   MXNDM  INPOL  NPNT  NPNTTS  IAVM  ISAM
2     SPH_1_X       SPH_1_Y       SPH_1_Z
3       ...           ...           ...
4   SPH_NSPH_X    SPH_NSPH_Y    SPH_NSPH_Z
5 IN_TH_FI  IN_TH_ST IN_TH_LA SC_TH_FI  SC_TH_ST SC_TH_LA
6 IN_PH_FI  IN_PH_ST IN_PH_LA SC_PH_FI  SC_PH_ST SC_PH_LA
7   JWTM
8  0
9 # [RUNTIME OPTIONS]
10 ...
```

The main differences with respect to the DSPH format presented in Section 2.4.1 are restricted to the first lines. Specifically, contains 9 parameters, instead of 6, with the following roles:

- NSPH: the number of spheres in the aggregate (integer, must be equal to that given in DEDFB)

- LI: the maximum multipolar field internal expansion order (integer).

- LE: the maximum multipolar field external expansion order (integer).

- MXNDM: the T-matrix leading dimension, to be set as MXNDM $=$ 2 NSPH$\cdot$LI$\cdot$(LI+2) (integer, used only by the *FORTRAN* code).

- INPOL: the incident radiation field polarization state (integer, '0' for linear, '1' for circular).

- NPNT: number of integration points for non-transition layers (integer).

- NPNTTS: number of integration points for transition layers (integer).

- IAVM: flag for averaging modes (integer, '0' computes averages for amplitudes and cross-sections, '1' computes averages for amplitudes, cross-sections and intensity).

- ISAM: flag for geometry reference plane (integer, '0' for scattering plane, '1' for scattering plane across z-axis, > 1 for scattering plane across z-axis and fixed scattering angle).

The following lines, here ranging from line 2 to line 4 are used to provide the spatial coordinates of each of the NSPH spheres in the aggregate, in the form of one (x, y, z) Cartesian triplet per line, with respect to a reference frame conventionally defined with the z-axis oriented along the radiation propagation direction. An important caveat in the definition of the sphere aggregate geometry is that **the spherical monomers cannot overlap in space**, since this breaks down the theoretical framework at the basis of the calculations.

The remaining lines, from line 5 to 10 have the same format and meaning as lines from 2 to 7 in DSPH and they provide, respectively, the azimuth angle settings (initial value, increment and final value) for the incident and scattered radiation (line 5), the elevation angle settings for incident and scattered radiation (6), the index of the wavelength to optionally save a T-matrix (JWTM in 7), an end-of-data code ('0' in line 8), plus additional optional runtime parameters (lines 9 and beyond).

2.5.2 Output files

The output of the np_cluster application is formally similar to the one produced by np_sphere, described in Section 2.4.2. For np_cluster, the output file naming scheme is:

- c_OEDFB: a plain text file containing a log of the configuration.

- c_OCLU: the main result file in plain text format.

- c_OCLU.hd5: the same results, but stored in HDF5 binary format.

- c_TEDF: a proprietary binary file storing configuration information.

- `c_TEDF.hd5`: the same configuration information stored in `HDF5` format.

- `c_TPPOAN`: a proprietary binary file with the main results (deprecated).

Also in this case, the T-matrix computed for a specific wavelength can be saved by setting the `JWTM` to some value other than '0', producing a binary `HDF5` file named `c_TTMS.hdf5` and its equivalent proprietary binary `c_TTMS`. All the output files are written under the folder pointed to by the `OUTPUT_DIR` command line argument (silently overwriting previous results that were not renamed, if present).

The `c_OCLU` plain text file contains all the code results with the following format (again, text marked with the hash character '#' is for commenting purposes and does not appear in the actual files, while backslash characters '\' imply line continuation):

```
## Preamble: replication of input configuration ##
 READ(IR,*)NSPH,LI,LE,MXNDM,INPOL,NPNT,NPNTTS,IAVM,ISAM
  nsph  li  le  (2*nsph*li*(li+2))  inpol  npnt  npntts  iavm  isam
 READ(IR,*)RXX(I),RYY(I),RZZ(I)
    sph_1_x       sph_1_y       sph_1_z
      ...           ...           ...
  sph_nsph_x    sph_nsph_y    sph_nsph_z
READ(IR,*)TH,THSTP,THLST,THS,THSSTP,THSLST
 th  thstp  thlst  ths thsstp  thslst
READ(IR,*)PH,PHSTP,PHLST,PHS,PHSSTP,PHSLST
 ph  phstp  phlst  phs  phsstp  phslst
 READ(IR,*)JWTM
  jwtm
  READ(ITIN)NSPHT
  READ(ITIN)(IOG(I),I=1,NSPH)
  READ(ITIN)EXDC,WP,XIP,IDFC,NXI
  READ(ITIN)(XIV(I),I=1,NXI)
  READ(ITIN)NSHL(I),ROS(I)
  READ(ITIN)(RCF(I,NS),NS=1,NSH)

  REFR. INDEX OF EXTERNAL MEDIUM= exdc^2
# The following line only appears if XITYP = 1
  VK=ref_vacuum_k, XI IS SCALE FACTOR FOR LENGTHS

# The following section repeats for all wavelength scales
```

```
========== JXI =  1 ====================
# If XITYP = 1:
  XI=xi_value
# otherwise:
  VK=2*pi/lambda, XI=ang_freq/ref_ang_freq
   LIN|CIRC # depending on the value of INPOL
# The following block repeats for every sphere type
     SPHERE   1
# If the sphere has only one layer
  SIZE=2*pi/lambda*rsph, REFRACTIVE INDEX=Real(n) Imag(n)
# otherwise:
  SIZE=2*pi/lambda*rsph
# Section of average quantities for single sphere types
# Integrated cross-sections and albedoes
 ----- SCS ----- ABS ----- EXS ----- ALBEDS --
 scattering_cs absorption_cs extinction_cs albedo
# Ratios of cross-sections to geometrical sections
 ---- SCS/GS -- ABS/GS -- EXS/GS ---
 scattering_cs/gs absorption_cs/gs extinction_cs/gs
 FSAS = real_part  imag_part # forward scattering amplitude
 QSCHU=4*pi*Imag(FSAS)/GS, PSCHU=4*pi*Real(FSAS)/GS, SOMAG= \
   Abs(FSAS)/(4*pi*k^3) # k is the wave vector
# Asymmetry parameter and radiation pressure cross-section
  COSAV=asymmetry_parameter, RAPRS=radiation_pressure_cs
# Cross-section contributions to extinction and scattering torque
# components along radiation propagation vector for polarization
# state 1 (parallel to polarization plane in linear polarization
# or co-rotating with polarization vector for circular polarization.
  IPO= 1, TQEk=ext_torque_k_cs, TQSk=scat_torque_k_cs
# Cross-section contributions to extinction and scattering torque
# components along radiation propagation vector for polarization
# state 1 (perpendicular to polarization plane in linear polarization
# or counter-rotating with polarization vector for circular polarization.
  IPO= 2, TQEk=ext_torque_k_cs, TQSk=scat_torque_k_cs
... # other SPHERE blocks up to SPHERE  ID_NSPH ...
# Section for average quantities of the whole aggregate
# Total forward scattering amplitude
  FSAT= Re(FSAT)  Im(FSAT)
  QSCHU=4*pi*Imag(FSAT)/TGS, PSCHU=4*pi*Real(FSAT)/TGS, SOMAG= \
   Abs(FSAT)/(4*pi*k^3) # k is the wave vector
# Section for average quantities of the whole aggregate
```

```
     CLUSTER (ENSEMBLE AVERAGE, MODE iavm)
# Average quantities are divided by polarization state. For
# linearly polarized incident radiation, the parallel state is
# presented first and the perpendicular state comes second. If
# the incident radiation is circularly polarized, instead, the
# co-rotating state is presented first and the counter-rotating
# one comes second.
   LIN|CIRC -1 # depending on INPOL, parallel / co-rotating
# Direction-averaged aggregate cross-sections and albedoes
 ----- SCC ----- ABC ----- EXC ----- ALBEDC --
 scattering_cs  absorption_cs  extinction_cs  albedo
# Ratios of cross-sections to total geometric cross-section (computed
# as the sum of the geometric cross-sections of all the spheres in the
# aggregate)
 --- SCC/TGS - ABC/TGS - EXC/TGS ---
scattering_cs/tgs  absorption_cs/tgs  extinction_cs/tgs
# Ratios of the aggregate cross-sections over the sum of the single
# spheres cross-sections
 ---- SCCRT --- ABCRT --- EXCRT ----
  scattering_cs/sum_of_scattering_cs  absorption_cs/sum_of_absorption_cs \
  extrinction_cs/sum_of_extinction_cs
# Complex integrated scattering amplitude of the aggregate: fsac11 is
# for polarization parallel to incidence; fsac21 is for polarization
# perpendicular to incidence
  FSAC(1,1)=  Re(fsac11)  Im(fsac11)  FSAC(2,1)=  Re(fsac21)  Im(fsac21)
# Ratios of the real and imaginary parts of the aggregate's forward
# scattering amplitudes for the above polarization states with respect
# to the sum of the single spheres forward scattering amplitudes
  RE(FSAC(1,1))/RE(TFSAS)= Re(fsac11)/Re(fsat), IM(FSAC(1,1))/IM(TFSAS)= \
    Im(fsac11)/Im(fsat)
  QSCHU=4*pi*Im(fsac11)/Im(fsat), PSCHU=4*pi*Re(fsac11)/Re(fsat), SOMAG= \
    Abs(fsac11)/(4*pi*k^3) # k is the wave vector
# Asymmetry parameter and radiation pressure cross-section (cfr. Borghese
# et al. 2007, Sec. 3.2.1)
  COSAV=asymmetry_parameter, RAPRS=radiation_pressure_cs
# Cross-section for the force along radiation incidence direction
  Fk=  f_k_cs
   LIN|CIRC  1 # depending on INPOL, perpendicular / counter-rotating
... # repeats the same information for the different polarization state
    # up to the ---- SCCRT --- ABCRT --- EXCRT ---- section
# Complex integrated scattering amplitude of the aggregate: fsac22 is
```

```
# for polarization parallel to incidence; fsac12 is for polarization
# perpendicular to incidence
  FSAC(2,2)=  Re(fsac22)  Im(fsac22)  FSAC(1,2)=  Re(fsac12)  Im(fsac12)
# Ratios of the real and imaginary parts of the aggregate's forward
# scattering amplitudes for the above polarization states with respect
# to the sum of the single spheres forward scattering amplitudes
  RE(FSAC(2,2))/RE(TFSAS)= Re(fsac22)/Re(fsat), IM(FSAC(2,2))/IM(TFSAS)= \
    Im(fsac22)/Im(fsat)
  QSCHU=4*pi*Im(fsac22)/Im(fsat), PSCHU=4*pi*Re(fsac22)/Re(fsat), SOMAG= \
    Abs(fsac22)/(4*pi*k^3) # k is the wave vector
# Asymmetry parameter and radiation pressure cross-section (cfr. Borghese
# et al. 2007, Sec. 3.2.1)
  COSAV=asymmetry_parameter, RAPRS=radiation_pressure_cs
# Cross-section for the force along radiation incidence direction
  Fk=force_on_prop_cs
# Averaged optical asymmetry factors, evaluated as normalized differences
# of forward scattering amplitudes between polarization states
  (RE(FSAC(1,1))-RE(FSAC(2,2)))/RE(FSAC(1,1))=normalized_real_asymmetry
  (IM(FSAC(1,1))-IM(FSAC(2,2)))/IM(FSAC(1,1))=normalized_imag_asymmetry
# Differential values (repeats for each scattering angle)
********* JTH =  1, JPH =  1, JTHS =  1, JPHS =  1 ***********
# Incidence and scattering directions
  TIDG=inc_theta, PIDG=inc_phi, TSDG=scat_theta, PSDG=scat_phi
  SCAND=scattering_angle # all angles in degrees
# Control parameters on incidence and scattering directions
# referred to meridional plane (cfr. Borghese et al. 2007,
# Sec. 2.7)
  CFMP=inc_cpar_mp, SFMP=sca_cpar_mp
# Control parameters on incidence and scattering directions
# referred to scattering plane (cfr. Borghese et al. 2007,
# Sec. 2.7)
  CFSP=inc_cpar_sp, SFSP=sca_cpar_sp
# x,y,z components of unitary vector perpendicular to incidence plane
  UNI=(uni_x, uni_y, uni_z)
# x,y,z components of unitary vector perpendicular to scattering plane
  UNS=(uns_x, uns_y, uns_z)
   LIN|CIRC # depending on INPOL
# Scattering amplitudes and Mueller transformation matrices for each
# sphere type
     SPHERE  1
# Complex differential scattering amplitude of the sphere
```

```
# along scattering direction: sas11 has the same polarization
# of the parallel incident field; sas21 has perpendicular
# polarization to the parallel incident field
  SAS(1,1)= Re(sas11) Im(sas11), SAS(2,1)= Re(sas21) Im(sas21)
# sas12 has perpendicular polarization to the perpendicular
# incident field; sas22 has the same polarization of the
# perpendicular incident field
  SAS(1,2)= Re(sas12) Im(sas12), SAS(2,2)= Re(sas22) Im(sas22)
# x,y,z components of the radiation forces exerted on sphere
  Fx=   rad_force_x, Fy=   rad_force_y, Fz=   rad_force_z
# Mueller transformation matrix for connection of Stokes
# parameters from incident field to scattered one referred to
# meridional plane (phase matrix, cfr. Borghese et al. 2007,
# Sec. 2.8.1)
  MULS
         MULS11    MULS12    MULS13    MULS14
         MULS21    MULS22    MULS23    MULS24
         MULS31    MULS32    MULS33    MULS34
         MULS41    MULS42    MULS43    MULS44
# Mueller transformation matrix for connection of Stokes
# parameters from incident field to scattered one referred to
# scattering plane (scattering matrix, cfr. Borghese et al.
# 2007, Sec. 2.8.1)
  MULSLR
         MULSLR11  MULSLR12  MULSLR13  MULSLR14
         MULSLR21  MULSLR22  MULSLR23  MULSLR24
         MULSLR31  MULSLR32  MULSLR33  MULSLR34
         MULSLR41  MULSLR42  MULSLR43  MULSLR44
  ... ... ... # repeats for all sphere types
# Sums of the scattering amplitudes of all the spheres for the
# current scattering direction. sat11 has the same polarization
# of the parallel incident field; sat21 has perpendicular
# polarization with respect to the parallel incident field; sat12
# has perpendicular polarization to the perpendicular incident
# field; sat22 has the same polarization of the perpendicular
# incident field
  SAT(1,1)=  Re(sat11)  Im(sat11), SAT(2,1)=  Re(sat21)  Im(sat21)
  SAT(1,2)=  Re(sat12)  Im(sat12), SAT(2,2)=  Re(sat22)  Im(sat22)
# Section for the differential aggregate values
     CLUSTER
# Differential quantities are divided by polarization state. For
```

```
# linearly polarized incident radiation, the parallel state is
# presented first and the perpendicular state comes second. If
# the incident radiation is circularly polarized, instead, the
# co-rotating state is presented first and the counter-rotating
# one comes second.
   LIN|CIRC -1 # depending on INPOL, parallel / co-rotating
# Differential aggregate cross-sections and albedoes for current direction
 ----- SCC ----- ABC ----- EXC ----- ALBEDC --
 scattering_cs  absorption_cs  extinction_cs  albedo
# Ratios of cross-sections to total geometric cross-section (computed
# as the sum of the geometric cross-sections of all the spheres in the
# aggregate)
 --- SCC/TGS - ABC/TGS - EXC/TGS ---
scattering_cs/tgs  absorption_cs/tgs  extinction_cs/tgs
# Ratios of the aggregate cross-sections over the sum of the single
# spheres cross-sections
 ---- SCCRT --- ABCRT --- EXCRT ----
  scattering_cs/sum_of_scattering_cs  absorption_cs/sum_of_absorption_cs \
  extrinction_cs/sum_of_extinction_cs
# Complex integrated scattering amplitude of the aggregate: fsac11 is
# for polarization parallel to incidence; fsac21 is for polarization
# perpendicular to incidence
  FSAC(1,1)=  Re(fsac11)  Im(fsac11)  FSAC(2,1)=  Re(fsac21)  Im(fsac21)
# Ratios of the real and imaginary parts of the aggregate's forward
# scattering amplitudes for the above polarization states with respect
# to the sum of the single spheres forward scattering amplitudes
  RE(FSAC(1,1))/RE(TFSAS)= Re(fsac11)/Re(fsat), IM(FSAC(1,1))/IM(TFSAS)= \
    Im(fsac11)/Im(fsat)
  QSCHU=4*pi*Im(fsac11)/Im(fsat), PSCHU=4*pi*Re(fsac11)/Re(fsat), SOMAG= \
    Abs(fsac11)/(4*pi*k^3) # k is the wave vector
# Asymmetry parameter and radiation pressure cross-section (cfr. Borghese
# et al. 2007, Sec. 3.2.1)
  COSAV=asymmetry_parameter, RAPRS=radiation_pressure_cs
# Radiation force cross-sections along the polarization direction,
# perpendicular to the polarization direction and along the radiation
# propagation direction
  Fl=force_on_pol_cs, Fr=force_cros_pol_cs, Fk=force_on_prop_cs
# Radiation force cross-sections for the Cartesian force components
  Fx=force_along_x_cs, Fy=force_along_y_cs, Fz=force_along_z_cs
# Extinction contribution to torque cross-sections along the polarization
# direction, perpendicular to the polarization direction and along the
```

```
# radiation propagation direction
  TQEl=ex_tor_on_pol_cs, TQEr=ex_tor_cros_pol_cs, TQEk=ex_tor_on_prop_cs
# Scattering contribution to torque cross-sections along the polarization
# direction, perpendicular to the polarization direction and along the
# radiation propagation direction
  TQSl=sc_tor_on_pol_cs, TQSr=sc_tor_cros_pol_cs, TQSk=sc_tor_on_prop_cs
# Extinction contribution to torque cross-sections for the Cartesian
# torque components
  TQEx=ex_tor_along_x_cs, TQEy=ex_tor_along_y_cs, TQEk=ex_tor_along_z_cs
# Scattering contribution to torque cross-sections for the Cartesian
# torque components
  TQSx=sc_tor_along_x_cs, TQSy=sc_tor_along_y_cs, TQSz=sc_tor_along_z_cs
   LIN|CIRC  1 # depending on INPOL, perpendicular / counter-rotating
  ... # repeats the same information for the different polarization state
      # up to the torque cross-section components
# Differential optical asymmetry factors, evaluated as normalized
# differences of forward scattering amplitudes between polarization states
  (RE(FSAC(1,1))-RE(FSAC(2,2)))/RE(FSAC(1,1))=normalized_real_asymmetry
  (IM(FSAC(1,1))-IM(FSAC(2,2)))/IM(FSAC(1,1))=normalized_imag_asymmetry
# Mueller transformation matrix for connection of Stokes parameters of
# the whole aggregate from incident field to scattered one referred to
# meridional plane (phase matrix, cfr. Borghese et al. 2007, Sec. 2.8.1)
  MULC
         MULC11    MULC12    MULC13    MULC14
         MULC21    MULC22    MULC23    MULC24
         MULC31    MULC32    MULC33    MULC34
         MULC41    MULC42    MULC43    MULC44
# Mueller transformation matrix for connection of Stokes parameters of
# the whole aggregate from incident field to scattered one referred to
# scattering plane (scattering matrix, cfr. Borghese et al. 2007,
# Sec. 2.8.1)
  MULCLR
         MULCLR11  MULCLR12  MULCLR13  MULCLR14
         MULCLR21  MULCLR22  MULCLR23  MULCLR24
         MULCLR31  MULCLR32  MULCLR33  MULCLR34
         MULCLR41  MULCLR42  MULCLR43  MULCLR44
  ... ... # other differential values for all scattering angles
... # other JXI blocks up to a total of NXI scales
```

Similarly to what happens with `np_sphere`, also `np_cluster` stores the computed values in an equivalent `HDF5` binary file that contains the same

information as the plain text output. The data are arranged the form of vectors of equivalent values, sorted in order of increasing wavelength and, if necessary, with the same sequence of computed directions.

2.6 np_inclusion

`np_inclusion` is formally similar to `np_cluster` in its implementation and input data. The output, on the other hand, presents some analogies with that of `np_sphere`, because the particle aggregate is eventually embedded in a single spherical outer coating. The execution scheme is the same of the other applications and it requires, as usual, a calculation description file, a geometry configuration with runtime options and the path to an existing output folder (where previously existing calculation results are silently overwritten).

2.6.1 Input files

The calculation description file for a spherical particle with an aggregate of internal inclusions is also conventionally named `DEDFB`. It has the following format:

```
NSPH  1
EXDC  WP  XIP  IDFC  NXI  INSTPC  XITYP
    XI_1
    ...
   XI_NXI
ID_1 ID_2 ... ID_NSPH
 NLYRS_1   RSPH_1
RLYR_1
 ... ... # repeats for all layers in sphere of type 1, plus 1
RLYR_NLYRS_1 + 1
 NLYRS_2  RSPH_2
RLYR_1
 ... ... # repeats for all layers in sphere of type 2
RLYR_NLYRS_2
... # repeats for all sphere types
RLYR_1
 ... ... # repeats for all layers in sphere of type ID_NSPH
```

```
 RLYR_NLYRS_ID_NSPH
(REAL_EPS_ID_1_LYR_1_XI_1,IMAG_EPS_ID_1_LYR_1_XI_1)
(REAL_EPS_ID_1_LYR_3_XI_1,IMAG_EPS_ID_1_LYR_3_XI_1)
... ... ... # repeating for all the odd layers of sphere type 1 plus one
(REAL_EPS_ID_1_LYR_NLYRS+1_XI_1,IMAG_EPS_ID_1_LYR_NLYRS+1_XI_1)
... ... # repeating for all sphere types
(REAL_EPS_ID_NSPH_LYR_1_XI_1,IMAG_EPS_ID_NSPH_LYR_1_XI_1)
(REAL_EPS_ID_NSPH_LYR_3_XI_1,IMAG_EPS_ID_NSPH_LYR_3_XI_1)
... ... ... # repeating for all the odd layers of sphere type ID_NSPH
(REAL_EPS_ID_NSPH_LYR_NLYRS_XI_1,IMAG_EPS_ID_NSPH_LYR_NLYRS_XI_1)
... # repeating for all wavelengths
(REAL_EPS_ID_1_LYR_1_XI_NXI,IMAG_EPS_ID_1_LYR_1_XI_NXI)
(REAL_EPS_ID_1_LYR_3_XI_NXI,IMAG_EPS_ID_1_LYR_3_XI_NXI)
... ... ... # repeating for all the odd layers of sphere type 1 plus one
(REAL_EPS_ID_1_LYR_NLYRS+1_XI_NXI,IMAG_EPS_ID_1_LYR_NLYRS+1_XI_NXI)
... ... # repeating for all sphere types
(REAL_EPS_ID_NSPH_LYR_1_XI_NXI,IMAG_EPS_ID_NSPH_LYR_1_XI_NXI)
(REAL_EPS_ID_NSPH_LYR_3_XI_NXI,IMAG_EPS_ID_NSPH_LYR_3_XI_NXI)
... ... ... # repeating for all the odd layers of sphere type ID_NSPH
(REAL_EPS_ID_NSPH_LYR_NLYRS_XI_NXI,IMAG_EPS_ID_NSPH_LYR_NLYRS_XI_NXI)
0
```

Lines from 1 to 6 in the `DEDFB` file for `np_inclusion` have the same meaning and format as the ones used for `np_cluster`, discussed in Section 2.5.1. The only difference is that the flag to enable an external spherical coating, after `NSPH` in line 1, must be set to '1' instead of '0' here. Lines from 7 to 18 describe the fractional radii (i.e. the fraction of the actual sphere radii) that are covered by the the outer shell of each layer in each type of sphere, with the caveat that the first sphere must specify an additional layer, acting as the external coating of the whole particle. This additional layer has two special properties, with respect the other layers of the different sphere types: it has *no transition layer*, because it acts as coating for the whole aggregate and not just the first sphere, and it is always centered at the origin of the coordinate system, even if the first sphere is spatially offset. This layer is the only one that has a fractional radius larger than unity and **it must fully include** all the other spheres of the aggregate. The last part of the configuration, from line 19 to 37, contains the declaration of optical constants of the materials used in all the odd layers of each type of sphere, with the caveat

that the coating layer must be declared as an additional layer for the first type of sphere. The '0' character appearing in line 38 is the usual end-of-file code.

Similarly to the case of the previous applications, `np_inclusion` needs a geometry configuration file, whose format is identical to the one used by `np_cluster`. For this case, the file is conventionally named `DINCLU` and its structure is as follows:

```
1  NSPH   LI   LE   MXNDM  INPOL  NPNT  NPNTTS  IAVM  ISAM
2     SPH_1_X       SPH_1_Y       SPH_1_Z
3       ...           ...           ...
4   SPH_NSPH_X    SPH_NSPH_Y    SPH_NSPH_Z
5 IN_TH_FI  IN_TH_ST IN_TH_LA SC_TH_FI  SC_TH_ST SC_TH_LA
6 IN_PH_FI  IN_PH_ST IN_PH_LA SC_PH_FI  SC_PH_ST SC_PH_LA
7   JWTM
8  0
9 # [RUNTIME OPTIONS]
10 ...
```

The meaning of the parameters provided in a `DINCLU` file is exactly the same as the one of those given in a `DCLU` file and, therefore, discussed in Section 2.5.1 for the case of `np_cluster`. `np_inclusion` has the same limitation concerning the fact that the various spheres in the aggregate cannot overlap in space.

2.6.2 Output files

Execution of `np_inclusion` results in the production of the following output files under the destination folder:

- `c_OEDFB`: a plain text file containing a log of the configuration.

- `c_OINCLU`: the main result file in plain text format.

- `c_OINCLU.hd5`: the same results, but stored in `HDF5` binary format.

- `c_TEDF`: a proprietary binary file storing configuration information.

- `c_TEDF.hd5`: the same configuration information stored in `HDF5` format.

- `c_TPPOAN`: a proprietary binary file with the main results (deprecated).

As usual, setting a `JWTM` parameter to any value between '1' and '`NXI`', will result in the production of the T-matrix output for the corresponding wavelength, both in `HDF5` binary format (`c_TTMS.hd5`) and in the proprietary binary version (`c_TTMS`).

The structure of the plain text output produced by `np_inclusion` is as follows (as always, text marked with the hash character '`#`' is for commenting purposes and does not appear in the actual files, while backslash characters '\' imply line continuation):

```
## Preamble: replication of input configuration ##
 READ(IR,*)NSPH,LI,LE,MXNDM,INPOL,NPNT,NPNTTS,IAVM,ISAM
  nsph  li  le  (2*nsph*li*(li+2))  inpol  npnt  npntts  iavm  isam
 READ(IR,*)RXX(I),RYY(I),RZZ(I)
    sph_1_x       sph_1_y       sph_1_z
      ...           ...           ...
  sph_nsph_x    sph_nsph_y    sph_nsph_z
READ(IR,*)TH,THSTP,THLST,THS,THSSTP,THSLST
 th  thstp  thlst  ths thsstp  thslst
READ(IR,*)PH,PHSTP,PHLST,PHS,PHSSTP,PHSLST
 ph  phstp  phlst  phs  phsstp  phslst
 READ(IR,*)JWTM
  jwtm
  READ(ITIN)NSPHT
  READ(ITIN)(IOG(I),I=1,NSPH)
  READ(ITIN)EXDC,WP,XIP,IDFC,NXI
  READ(ITIN)(XIV(I),I=1,NXI)
  READ(ITIN)NSHL(I),ROS(I)
  READ(ITIN)(RCF(I,NS),NS=1,NSH)

  REFR. INDEX OF EXTERNAL MEDIUM= exdc^2
# The following line only appears if XITYP = 1
  VK=ref_vacuum_k, XI IS SCALE FACTOR FOR LENGTHS
# The following section repeats for all wavelength scales
========== JXI =  1 ====================
# If XITYP = 1:
  XI=xi_value
# otherwise:
  VK=2*pi/lambda, XI=ang_freq/ref_ang_freq
# If SPHERE 1 has only two layers (one layer + external coating)
```

```
  SPHERE N.  1: SIZE=2*pi/lambda*rsph, REFRACTIVE INDEX=Real(n) Imag(n)
# otherwise:
  SPHERE N.  1: SIZE=2*pi/lambda*rsph
  ... # repeats for all types of sphere
# If SPHERE ID_NSPH has only one layer
  SPHERE N. ID_NSPH: SIZE=2*pi/lambda*rsph, REFRACTIVE INDEX=Real(n) \
    Imag(n)
# otherwise:
  SPHERE N. ID_NSPH: SIZE=2*pi/lambda*rsph
# Section for direction-averaged values
     ENSEMBLE AVERAGE, MODE iavm
# Average quantities are divided by polarization state. For
# linearly polarized incident radiation, the parallel state is
# presented first and the perpendicular state comes second. If
# the incident radiation is circularly polarized, instead, the
# co-rotating state is presented first and the counter-rotating
# one comes second.
   LIN|CIRC -1 # depending on INPOL, parallel / co-rotating
# Direction-averaged aggregate cross-sections and albedoes
 ----- SCS ----- ABS ----- EXS ----- ALBEDS --
scattering_cs absorption_cs extinction_cs albedo
# Ratios of cross-sections to geometric cross-section (computed
# as the geometric cross-sections of external spherical coating)
 ---- SCS/GS -- ABS/GS -- EXS/GS ---
scattering_cs/gs  absorption_cs/gs  extinction_cs/gs
# Complex integrated forward scattering amplitudes of the aggregate:
# fsas11 is for polarization parallel to incidence; fsas21 is for
# polarization perpendicular to incidence
  FSAS(1,1)= Re(fsas11) Im(fsas11) FSAS(2,1)= Re(fsas21) Im(fsas21)
  QSCHU=4*pi*Im(fsas11)/Im(fsas), PSCHU=4*pi*Re(fsas11)/Re(fsas), SOMAG= \
    Abs(fsas11)/(4*pi*k^3) # k is the wave vector
# Asymmetry parameter and radiation pressure cross-section (cfr. Borghese
# et al. 2007, Sec. 3.2.1)
COSAV=asymmetry_parameter, RAPRS=radiation_pressure_cs
# Radiation force cross-sections along the radiation propagation direction
  Fk=force_on_prop_cs
# The same information is given for the other polarization state
   LIN|CIRC  1 # depending on INPOL, perpendicular / counter-rotating
  ... # same fields as for LIN -1
# Averaged optical asymmetry factors, evaluated as normalized differences
# of forward scattering amplitudes between polarization states
```

```
  (RE(FSAS(1,1))-RE(FSAC(S,2)))/RE(FSAS(1,1))=normalized_real_asymmetry
  (IM(FSAS(1,1))-IM(FSAC(S,2)))/IM(FSAS(1,1))=normalized_imag_asymmetry
# Differential values (repeats for each scattering angle)
********* JTH =  1, JPH =  1, JTHS =  1, JPHS =  1 ***********
# Incidence and scattering directions
  TIDG=inc_theta, PIDG=inc_phi, TSDG=scat_theta, PSDG=scat_phi
  SCAND=scattering_angle # all angles in degrees
# Control parameters on incidence and scattering directions
# referred to meridional plane (cfr. Borghese et al. 2007,
# Sec. 2.7)
  CFMP=inc_cpar_mp, SFMP=sca_cpar_mp
# Control parameters on incidence and scattering directions
# referred to scattering plane (cfr. Borghese et al. 2007,
# Sec. 2.7)
  CFSP=inc_cpar_sp, SFSP=sca_cpar_sp
# x,y,z components of unitary vector perpendicular to incidence plane
  UNI=(uni_x, uni_y, uni_z)
# x,y,z components of unitary vector perpendicular to scattering plane
  UNS=(uns_x, uns_y, uns_z)
     SINGLE SCATTERER
# Differential quantities are divided by polarization state. For
# linearly polarized incident radiation, the parallel state is
# presented first and the perpendicular state comes second. If
# the incident radiation is circularly polarized, instead, the
# co-rotating state is presented first and the counter-rotating
# one comes second.
   LIN|CIRC -1 # depending on INPOL, parallel / co-rotating
# Differential aggregate cross-sections and albedoes
 ----- SCS ----- ABS ----- EXS ----- ALBEDS --
  scattering_cs absorption_cs extinction_cs albedo
# Ratios of cross-sections to geometric cross-section (computed
# as the geometric cross-sections of external spherical coating)
 ---- SCS/GS -- ABS/GS -- EXS/GS ---
  scattering_cs/gs absorption_cs/gs extinction_cs/gs
# Complex differential forward scattering amplitude of the aggregate:
# fsas11 is for polarization parallel to incidence; fsas21 is for
# polarization perpendicular to incidence
  FSAS(1,1)= Re(fsas11) Im(fsas11) FSAS(2,1)= Re(fsas21) Im(fsas21)
# Complex differential scattering amplitude of the aggregate: sas11
# is for polarization parallel to incidence; sas21 is for polarization
# perpendicular to incidence
```

```
   SAS(1,1)=  Re(sas11)  Im(sas11)  SAS(2,1)=  Re(sas21)  Im(sas21)
  QSCHU=4*pi*Im(fsas11)/Im(fsas), PSCHU=4*pi*Re(fsas11)/Re(fsas), SOMAG= \
Abs(fsas11)/(4*pi*k^3) # k is the wave vector
# Asymmetry parameter and radiation pressure cross-section (cfr. Borghese
# et al. 2007, Sec. 3.2.1)
COSAV=asymmetry_parameter, RAPRS=radiation_pressure_cs
# Radiation force cross-sections along the polarization direction,
# perpendicular to the polarization direction and along the radiation
# propagation direction
  Fl=force_on_pol_cs, Fr=force_cros_pol_cs, Fk=force_on_prop_cs
# Radiation force cross-sections for the Cartesian force components
  Fx=force_along_x_cs, Fy=force_along_y_cs, Fz=force_along_z_cs
# Extinction contribution to torque cross-sections along the polarization
# direction, perpendicular to the polarization direction and along the
# radiation propagation direction
  TQEl=ex_tor_on_pol_cs, TQEr=ex_tor_cros_pol_cs, TQEk=ex_tor_on_prop_cs
# Scattering contribution to torque cross-sections along the polarization
# direction, perpendicular to the polarization direction and along the
# radiation propagation direction
  TQSl=sc_tor_on_pol_cs, TQSr=sc_tor_cros_pol_cs, TQSk=sc_tor_on_prop_cs
# Extinction contribution to torque cross-sections for the Cartesian
# torque components
  TQEx=ex_tor_along_x_cs, TQEy=ex_tor_along_y_cs, TQEk=ex_tor_along_z_cs
# Scattering contribution to torque cross-sections for the Cartesian
# torque components
  TQSx=sc_tor_along_x_cs, TQSy=sc_tor_along_y_cs, TQSz=sc_tor_along_z_cs
  LIN|CIRC 1 # depending on INPOL, perpendicular / counter-rotating
  ... # repeats the same information for the different polarization state
      # up to the torque cross-section components
# Differential optical asymmetry factors, evaluated as normalized
# differences of forward scattering amplitudes between polarization states
  (RE(FSAS(1,1))-RE(FSAS(2,2)))/RE(FSAS(1,1))=normalized_real_asymmetry
  (IM(FSAS(1,1))-IM(FSAS(2,2)))/IM(FSAS(1,1))=normalized_imag_asymmetry
# Mueller transformation matrix for connection of Stokes parameters of
# the whole aggregate from incident field to scattered one referred to
# meridional plane (phase matrix, cfr. Borghese et al. 2007, Sec. 2.8.1)
  MULL
         MULL11    MULL12    MULL13    MULL14
         MULL21    MULL22    MULL23    MULL24
         MULL31    MULL32    MULL33    MULL34
         MULL41    MULL42    MULL43    MULL44
```

```
# Mueller transformation matrix for connection of Stokes parameters of
# the whole aggregate from incident field to scattered one referred to
# scattering plane (scattering matrix, cfr. Borghese et al. 2007,
# Sec. 2.8.1)
  MULLLR
         MULLLR11  MULLLR12  MULLLR13  MULLLR14
         MULLLR21  MULLLR22  MULLLR23  MULLLR24
         MULLLR31  MULLLR32  MULLLR33  MULLLR34
         MULLLR41  MULLLR42  MULLLR43  MULLLR44
  ... ... # other differential values for all scattering angles
... # other JXI blocks up to a total of NXI scales
```

Also `np_inclusion`, like `np_sphere` and `np_cluster`, writes the same output information in a `HDF5` binary file named `c_OINCLU.hd5`, which, as usual, contains a set of vectors of equivalent values, arranged with the same scheme of wavelengths and directions as the one used in the plain text results file.

2.7 np_trapping

Although using the same input and output scheme of the other applications, `np_trapping` is inherently different because it solves for the forces and torques exerted on a particle, whose T-matrix has already been computed at a specific wavelength, when it interacts with a radiation field of that same wavelength. Therefore, instead of a scattering configuration file and a geometry configuration file, `np_trapping` requires a radiation beam description file and a file to control the calculation grid, together with optional runtime settings. The pre-computed T-matrix file must be a `HDF5 c_TTMS.hd5` binary file, such as the one computed by one of `np_sphere`, `np_cluster`, or `np_inclusion`, with `JWTM` set for the proper wavelength scale, and made available in the working directory where `np_trapping` is launched from.

Since `np_trapping` uses an iterative process to derive its solution, partial calculations can also be performed and subsequently used as a starting point for higher order extensions. The output of `np_trapping` consists of the forces and torques that the particle would experience if located in a grid of points within the radiation beam.

2.7.1 Input files

`np_trapping` takes two input configuration files, a radiation beam configuration, conventionally named `DFRFME`, and a configuration file for the result grid and any possible runtime options. As for the other applications, runtime settings are only handled by the *C++* implementation and ignored by the *FORTRAN 66* applications. The structure of the beam configuration file is:

```
1 L_START  L_END
2 LMODE  LM  NKSH  NRSH  NXSH  NYSH  NZSH  WLENFR
3 AN  FF  TRA
4 SPD  SPDFR  EXDCL
5 e  IXI
6 0
```

The parameters listed in line 1 have the following meaning:

- `L_START`: starting multipolar expansion order for the calculation (integer, set to '1' to start from the beginning, or to the order the order immediately following the last computed one, if extending a previous calculation).

- `L_END`: last multipolar expansion order to be considered in the calculation (integer, if smaller than 2·`LM`·(`LM`+2), with `LM` taken from line 2, produces partial output, otherwise gives final results).

Line 2, instead, contains the following parameters:

- `LMODE`: Laser mode flag (integer).

- `LM`: Maximum field expansion order of the input T-matrix (integer, must be set as the `LM` value used in `np_sphere`, or the `LE` value used in `np_cluster` / `np_inclusion`, depending on which application produced the `c_TTMS.hd5` T-matrix file).

- `NKSH`: number of wave vectors in half grid (integer).

- `NRSH`: number of spatial points along a grid half-axis (integer).

- `NRXH`: number of spatial points in half grid along x axis (integer).

- NRYH: number of spatial points in half grid along y axis (integer).

- NRZH: number of spatial points in half grid along z axis (integer).

- WLENFR: magnification factor (float).

The following line 3 contains the beaming lens parameters:

- AN: lens numerical aperture (float).

- FF: filling factor (float).

- TRA: transmission (float).

Line 4 contains parameters to describe the aberration plane:

- SPD: aberration plane distance (float).

- SPDFR: aberration plane distance multiplicative factor (float).

- EXDCL: dielectric constant of the suspension fluid (float).

Finally, lines 5 and 6 contain a flag character set to 'e', to signal that the calculation is intended for dielectric materials, the wavelength scale index IXI, pointing to the wavelength scale that was used to compute the T-matrix in the DEDFB file of the application that produced the c_TTMS.hd5 file and a '0' character used as a conventional end-of-file code.

The result grid and runtime options file used by np_trapping is named DLFFFT and it has the format:

```
1 JFT  0  JTW
2 0
3 [RUNTIME OPTIONS]
4 ...
```

where line 1 defines:

- JFT: forces and torques output flag (integer, if JFT = 0 writes forces and torques, if JFT < 0 writes only forces, if if JFT > 0 writes only torques).

- '0': a flag to ask output from all the computed grid points.

- JTW: a flag to produce formatted output files ('1' enables binary and formatted output, '0' produces only binary output).

The '0' character in line 2 serves as an end-of-data code.

2.7.2 Output files

The output of `np_trapping` consists of the follwoing files:

- `c_grid_scale.txt`: a plain ASCII file that provides a list of grid spacing layers around the beam focal plane (in meters).

- `c_OFRFME`: a plain ASCII file containing messages concerning the radiation field calculation and instructions on how to possibly expand to higher orders.

- `c_force_cs.txt`: a plain ASCII file containing the radiation field force components along the (x, y, z) reference frame exerted on the particle in every grid point.

- `c_torque_cs.txt`: a plain ASCII file containing the radiation field torque components along the (x, y, z) reference frame exerted on the particle in every grid point.

- `c_TLFFFT.hd5`: a `HDF5` binary file containing the force and torque cross-sections computed over the gird (the binary equivalent of `c_force_cs.txt` and `c_torque_cs.txt`).

- `c_TEMPTAPE1.hd5` & `c_TEMPTAPE2.hd5`: two `HD5` binary files containing temporary information to expand the calculation at higher orders.

- `c_TFRFME.hd5`: a `HD5` binary file containing the full solution of the radiation field, to be re-used if testing the same radiation beam on a different particle model.

The format of `c_force_cs.txt` and `c_torque_cs.txt` is:

```
1  IX_1  IY_1  IZ_1     CS_X_1_1_1     CS_Y_1_1_1     CS_Z_1_1_1
2              ... ... ... # repeats for all grid points along X
3  IX_NX IY_1  IZ_1    CS_X_NX_1_1    CS_Y_NX_1_1    CS_Z_NX_1_1
4        ... ... # repeats for all grid points along Y
5  IX_1  IY_NY  IZ_1   CS_X_1_NY_1    CS_Y_1_NY_1    CS_Z_1_NY_1
6              ... ... ... # repeats for all grid points along X
7  IX_NX IY_NY  IZ_1  CS_X_NX_NY_1   CS_Y_NX_NY_1  CS_Z_NX_NY1_1
8  ... # repeats for all grid points along Z
```

```
   IX_1  IY_1  IZ_NZ   CS_X_1_1_NZ    CS_Y_1_1_NZ    CS_Z_1_1_NZ
               ... ... ... # repeats for all grid points along X
   IX_NX IY_1  IZ_NZ  CS_X_NX_1_NZ   CS_Y_NX_1_NZ   CS_Z_NX_1_NZ
         ... ... # repeats for all grid points along Y
   IX_1  IY_NY IZ_NZ  CS_X_1_NY_NZ   CS_Y_1_NY_NZ   CS_Z_1_NY_NZ
               ... ... ... # repeats for all grid points along X
   IX_NX IY_NY IZ_NZ CS_X_NX_NY_NZ  CS_Y_NX_NY_NZ CS_Z_NX_NY1_NZ
```

Each line of the output file is composed by three integer values, identifying the position of the vertex on the grid, and by three floating point values, formatted with scientific notation, which, instead, express the cross-section contributions along the x, y, and z axes in units of m^2 respectively to the radiation force (in `c_force_cs.txt`) and to the torque (in `c_torque_cs.txt`).

2.8 Measurement units

As mentioned in Section 2.2, the `NP_TMcode` applications use the *International System* (IS) to define the units of measurement of input data and to produce the output cross-sections. This means that all references to wavelength are assumed to be expressed in meters (m) and all cross-sections are given in square meters (m^2). However, the solution of the electromagnetic equations that is computed internally uses the *CGS* system to handle the radiation fields. As a consequence, the cross-section contributions of the radiation forces and torques (which are expressed in meters) are related to the actual forces and torques exerted on the particle through the intensity of the incident radiation field.

To extract the radiation forces and torques from the cross-section contributions computed by `np_sphere`, `np_cluster`, `np_inclusion`, and `np_trapping`, some conversions need to be applied. To obtain the force from radiation pressure force cross-section, the quantities reported in the output are defined as:

$$\texttt{RAPR} = \frac{cF_{rad}}{nI_0} \qquad [\mathrm{m}^2], \tag{2.1}$$

where c is the speed of light in vacuum (in $\mathrm{m\,s}^{-1}$), F_{rad} is a radiation force (expressed in Newton), n is the external medium refractive index, and I_0 is the incident radiation field intensity (expressed in $\mathrm{J\,m}^{-2}\,\mathrm{s}^{-1}$. Therefore,

the expression of the force in *IS* units is determined by the value of I_0, by inverting Eq. (2.1):

$$F_{rad} = \frac{nI_0\texttt{RAPR}}{c} \quad [\mathrm{N}]. \tag{2.2}$$

The expression of the radiation field intensity is:

$$I_0 = (3 \times 10^4)^2 \frac{nc}{8\pi}|E_0|^2, \tag{2.3}$$

which, assuming $|E_0| = 1$, can be introduced in Eq. 2.2 to get:

$$F_{rad} = (3 \times 10^4)^2 \frac{n^2}{8\pi}\texttt{RAPR} \tag{2.4}$$

Similar considerations apply to the cross-section contributions of all force components (`Fx`, `Fy`, `Fz`, `Fr`, `Fl`, `Fk`).

Conversely, the values indicated as contributions of extinction and scattering cross-sections to torques are:

$$\texttt{TQvalue} = \frac{\Gamma c}{I_0} n k_v \quad [\mathrm{m}^2], \tag{2.5}$$

where Γ is the torque (in N·m) and k_v the the radiation wave number in vacuum ($k_v = 2\pi/\lambda$, where λ is the wavelength in m). Eq. (2.5) can be rearranged as:

$$\Gamma = \frac{I_0}{nck_v}\texttt{TQvalue}, \tag{2.6}$$

which, using again Eq. (2.3 with $|E_0| = 1$, leads to:

$$\Gamma = \frac{(3 \times 10^4)^2)}{8\pi k_v}\texttt{TQval} \qquad [\mathrm{N} \cdot \mathrm{m}]. \tag{2.7}$$

Chapter 3

Processing existing models

3.1 Modeling work-flow

The primary goal of the `NP_TMcode` applications is to estimate the cross-section of arbitrarily shaped particles for scattering, absorption, and extinction of electromagnetic radiation, as functions of wavelength. Additional effects, such as radiation forces and torques exerted on the model particles, can also be evaluated and, for well known radiation fields, such as those produced in controlled laboratory experiments, they can be used to predict the trapping conditions of sample particles.

This chapter describes the recommended process to run a sensible calculation, taking for granted that `NP_TMcode` was correctly installed. In order to allow for a full reproducibility of the steps illustrated in this chapter, the calculation procedure is shown for one of the development models that are included in the distributed code. The chosen example model represents a porphyrin nanotube, composed by 24 identical spherical monomers, like the one shown in Fig. 3.1. The data needed to test the procedure illustrated here are stored under the `test_data/cluster` and the `test_data/trapping` folders of the `NP_TMcode` software package. In the following, we shall make use of the following conventions:

- `INSTALL_DIR`: the file system root path of the `NP_TMcode` package (the folder containing the `COPYING` license file).

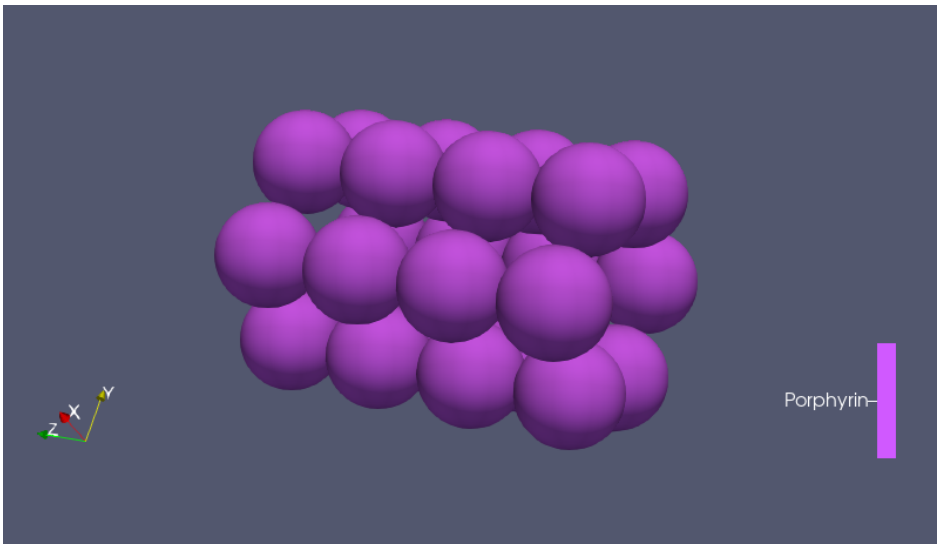

Figure 3.1: Representation of the 24-monomer porphyrin nanotube model particle used in the examples of this chapter. The size of the particle along its longest axis (corresponding to the reference frame's z-axis) is approximately $d \simeq 40\,\mathrm{nm}$.

- `WORK_DIR`: a working directory where the user has writing permissions.

- `USER`: the user name under the host system.

- `GROUP`: the group which the user is part of (for UNIX-like systems).

3.2 Preparing the data

`NP_TMcode` applications can be run from anywhere in the system and they can access any files whose absolute path is properly expanded. However, for the sake of simplicity and in order to easily maintain the results of multiple calculations, it is recommended to create a working directory and to operate from there. In a *bash*-complying *UNIX* shell, this can be achieved by:

```
$ mkdir WORK_DIR
$ cd WORK_DIR
```

The following step would be to copy (or link) the input data from the original folder in the working directory and to create a folder to store the results:

```
WORK_DIR$ cp INSTALL_DIR/test_data/cluster/D*_24 .
WORK_DIR$ mkdir cs_results
```

After executing the instructions above, a listing of the working directory should now show a screen like:

```
WORD_DIR$ ls -l
total 20
-rw-rw-r-- 1 USER GROUP  1365 nov 18 16:54 DCLU_24
-rw-rw-r-- 1 USER GROUP 11364 nov 18 16:54 DEDFB_24
drwxrwxr-x 2 USER GROUP  4096 nov 18 16:55 cs_results
```

If this is the situation, the data are ready for code execution, but some fine-tuning configuration is still necessary.

3.3 Choosing the expansion orders

As anticipated in Section 3.1, the test case is an aggregate of 24 spheres, representing a porphyrin nanotube, whose extinction properties are going to be evaluated for 181 wavelengths, in the range $420\,\text{nm} \leq \lambda \leq 600\,\text{nm}$ in steps of 1 nm. We are therefore going to use the `np_cluster` application, whose input files are described in Section 2.5.1.

Inspecting the `DEDFB_24` and the `DCLU_24` files, it turns out that the spherical monomers have radius $r_{sph} = 5\,\text{nm}$ and their aggregation results in a particle whose largest extension is approximately 38 nm (i.e., it takes a sphere with radius of at least 19 nm to wrap the particle inside). However, the first line of `DCLU_24` states that the code is configured to solve the particle using equal values of the inner and the outer expansion truncation orders (`LI`=`LE`= 2).[1] This might be too low for an accurate solution. One way to ascertain this is to use the *Wiscombe criterion* (Wiscombe, 1980). In our case, we are interested in estimating the proper way to configure the internal truncation order (suited to the dimensions of the monomers) and the external order (suited to the dimension of the whole particle). `NP_TMcode` provides a

[1] See the code technical description papers for a discussion of the truncation orders in the T-matrix formalism.

Python script, named `pywiscombe.py`, to perform these estimates. The way to execute it, for our case, is:

```
WORK_DIR$ INSTALL_DIR/src/scripts/pywiscombe.py --rad=5.0e-9 \
  --wave=4.1e-6 --li
Suggested truncation order is Lmax = 3
WORK_DIR$ INSTALL_DIR/src/scripts/pywiscombe.py --rad=19.0e-9 \
  --wave=4.1e-6 --le
Suggested truncation order is Lmax = 5
```

In this case, we executed the script twice: the first time setting the radius of a single monomer ($5{\cdot}10^{-9}$ m), the shortest wavelength to be computed ($4.1{\cdot}10^{-6}$ m) and asking for the recommended *internal* expansion order ("`--li`"), obtaining a value of 3; similarly, the second time we ran the script for the same wavelength, but for a sphere radius encircling the whole particle ($19.0{\cdot}10^{-6}$ m) and requesting the *external* order ("`--le`"), getting a recommendation of 5.

Recalling the explanation of the `np_cluster` input parameters of Section 2.5.1, we should therefore open the `DCLU_24` file and replace the first line:

```
1   24    2    2  5760   1  149  300    0    0
```

with:

```
1   24    3    5  5760   1  149  300    0    0
```

This will instruct the code to run with the recommended values for the internal truncation order `LI` and the external truncation order `LE`. We will also replace the second last line of `DCLU_24`, substituting:

```
28  1
```

with:

```
28  0
```

thus disabling, for the moment, the creation of the T-matrix output file.

3.4 Running the calculation

The solution of the scattering problem must be derived for all the wavelengths declared in the scattering configuration file `DEDFB`, for which we have

a definition of the optical properties of the material. NP_TMcode applications use various strategies to speed up such calculations, which involve simultaneous parallel solution of different wavelengths, leveraging of fast algebraic libraries, and offloading of algebraic operations to GPUs, if available. While the use of specialized libraries and the involvement of GPUs in the calculations are activated at the code compilation stage, the management of simultaneous wavelength solution can be controlled at runtime, by means of command line options and/or environment variables. np_sphere, np_cluster, and np_inclusion handle the wavelength parallelization via *MPI* multiprocessing and *OpenMP* multi-threading. Since our test case is a rather simple model, solved by np_cluster, we will use a single process to run the calculation, leveraging the thread hierarchy to perform parallel operations. If the code was built with *MPI* enabled, the recommended way to execute the calculation is:

```
WORK_DIR$ OMP_NUM_THREADS=NUM_WAVELEN_THREADS,NUM_WORK_THREADS \
  mpirun -n 1 INSTALL_DIR/build/cluster/np_cluster DEDFB_24 DCLU_24 \
  cs_results
```

otherwise, if *MPI* was not used, the command would be:

```
WORK_DIR$ OMP_NUM_THREADS=NUM_WAVELEN_THREADS,NUM_WORK_THREADS \
  INSTALL_DIR/build/cluster/np_cluster DEDFB_24 DCLU_24 cs_results
```

These commands use the OMP_NUM_THREADS environment variable to instruct np_cluster to split the available system threads in two hierarchical levels, solving for NUM_WAVELEN_THREADS simultaneous wavelengths per iteration and using NUM_WORK_THREADS work-sharing threads in each wavelength. The actual settings of the thread hierarchy depend on the complexity of the model and they must satisfy the constraint that the product of NUM_WAVELEN_THREADS and NUM_WORK_THREADS equals the total number of desired threads (arguably, not more than the available system threads). Simple models computed on many wavelengths may have NUM_WAVELEN_THREADS $>$ NUM_WORK_THREADS, while, for very complicated models, it is more advisable to set NUM_WAVELEN_THREADS $<$ NUM_WORK_THREADS. If *MPI* is used, a nearly equivalent way to run the code would be:

```
WORK_DIR$ OMP_NUM_THREADS=1,NUM_WORK_THREADS \
  mpirun -n NUM_WAVELEN_THREADS INSTALL_DIR/build/cluster/np_cluster \
```

```
  DEDFB_24 DCLU_24 cs_results
```

which is similar to the first method, but it uses *MPI processes* instead of threads.[2]

Taking as an example the execution of our test on the development hardware, namely an ASUS ZenBook with an Intel Core i9-13900H CPU supporting 20 threads and a NVIDIA GeForce RTX 4060 Ada Lovelace GPU with 8 GiB VRAM, the execution command is:[3]

```
1  WORK_DIR$ OMP_NUM_THREADS=5,4 INSTALL_DIR/build/cluster/np_cluster \
2    DEDFB_24 DCLU_24 cs_results
3  DEBUG: Proc-0 found 1 GPU device. # GPU detection message
4  INFO: Process 0 initializes MAGMA. # MAGMA detection message
5  INFO: making legacy configuration... done.
6  INFO: particle radius is 1.8708e-08.
7  INFO: iteration data requires 0.0094955GiB of RAM.
8  INFO: code execution needs 0.18991GiB of RAM.
9  INFO: Size of matrices to invert: 720 x 720.
10 INFO: using MAGMA calls.
11 Syncing OpenMP threads and starting the loop on wavelengths
12 ... # suppressed iteration logging information
13 INFO: finished scale iteration 181 of 181.
14 INFO: Closing thread-local output files of thread 0 and syncing threads.
15 INFO: Closing thread-local output files of thread 1 and syncing threads.
16 INFO: Closing thread-local output files of thread 3 and syncing threads.
17 INFO: Closing thread-local output files of thread 4 and syncing threads.
18 INFO: Closing thread-local output files of thread 2 and syncing threads.
19 INFO: Process 0 finalizes MAGMA.
20 INFO: Calculation lasted 9.514511s.
21 Finished: output written to cs_results/c_OCLU
```

While running, the code produces a set of log messages, written to `stdout`. The messages reported in lines between 3 and 10 concern the code configuration and the model characteristics (in terms of estimated particle radius and runtime memory requirements). From line 11 to line 13, the code produces

[2]For simple models like this test case, the two approaches only have small performance differences. For more complicated cases, detailed later, the use of *MPI* allows different processes to access multiple GPUs (if available), while all *OpenMP* threads are designed to work with the same device.

[3]Hash characters '#' are comments.

a verbose sequence of messages, reporting on the overall progress of the calculation. Finally, lines between 14 and 21 describe the application closing operations, summarize the computing time, and inform the user about the production of the calculation results.

3.5 Inspecting the results

Upon successful execution of the previous steps, a listing of the results folder should look like:

```
WORK_DIR$ ls -lh cs_results/
total 1,9M
-rw-rw-r-- 1 USER GROUP 1,1M nov 19 10:47 c_OCLU
-rw-rw-r-- 1 USER GROUP 393K nov 19 10:47 c_OCLU.hd5
-rw-rw-r-- 1 USER GROUP  19K nov 19 10:47 c_OEDFB
-rw-rw-r-- 1 USER GROUP 4,4K nov 19 10:47 c_TEDF
-rw-rw-r-- 1 USER GROUP  75K nov 19 10:47 c_TEDF.hd5
-rw-rw-r-- 1 USER GROUP   81 nov 19 10:47 c_timing_mpi0.log
-rw-rw-r-- 1 USER GROUP 271K nov 19 10:47 c_TPPOAN
```

As discussed in Section 2.5.2, the main calculation results are stored as formatted plain text in the `c_OCLU` file and, equivalently, in `HDF5` binary format in the `c_OCLU.hd5` file. The former can be inspected using any text editor, while the latter can be navigated by standard `HDF5` tools. However, since in most cases users are interested in the extraction of cross-sections as a function of wavelength, `NP_TMcode` provides some tools to quickly perform this operation. The main available instrument is the `parse_output.py` *Python* script, located in the `src/scripts` package folder. The correct way to obtain the particle cross-sections is to enter the results folder and to run the output parsing script from there:

```
WORK_DIR$ cd cs_results
cs_results$ INSTALL_DIR/src/scripts/parse_output.py --in c_OCLU --out
    cross --app=CLU
```

This command runs the script to parse the results stored in the `c_OCLU` file[4] (as specified by the "`-- in c_OCLU`" command line argument), writing its

[4]The output parsing script currently works on the plain text result file, while extension on the extraction of results from the `HDF5` file will be implemented later on.

output in a set of comma-separated values data files, named `cross_*.csv` (as implied by the "`--out cross`" argument). A listing of these files in the results folder will give:

```
cs_results$ ls -lh cross*
-rw-rw-r-- 1 USER GROUP  18K nov 19 11:43 cross_dcs1.csv
-rw-rw-r-- 1 USER GROUP  18K nov 19 11:43 cross_dcs2.csv
-rw-rw-r-- 1 USER GROUP  61K nov 19 11:43 cross_drp1.csv
-rw-rw-r-- 1 USER GROUP  61K nov 19 11:43 cross_drp2.csv
-rw-rw-r-- 1 USER GROUP  10K nov 19 11:43 cross_ics.csv
-rw-rw-r-- 1 USER GROUP 7,5K nov 19 11:43 cross_irp.csv
```

These new files contain tabulated values of the integrated cross-sections (expressed in m^2) as function of wavelength (expressed in m). The *integrated* scattering, absorption, and extinction *cross-sections* are collected in the file `cross_ics.csv`. Similarly, the *integrated radiation pressure* cross-sections (in m^2) are given in the `cross_irp.csv` file under a column named `RaPr`, together with the particle asymmetry parameters (named `CosAv` and expressed by pure numbers). The corresponding differential values are given in the files named `cross_dcs*.csv` and `cross_drp*.csv`, for each scattering direction and for two different polarization states ('1' being co-rotating and '2' being counter-rotating, since `INPOL` = 1 in `DCLU_24` defined a circularly polarized incident field).

A plot of the data extracted from the `cross_ics.csv` of our test case should reproduce the behavior illustrated in Fig. 3.2. Looking at the absorption, scattering and extinction cross-sections, we can appreciate that the modeled particle exhibits a prominent extinction peak, almost completely due to absorption, achieving a maximum at $\lambda = 494\,\text{nm}$, where the material has the largest imaginary part of its dielectric function, implying the strongest absorption of radiation.

3.6 Trapping calculations

To run a trapping calculation, we need the transition matrix of the particle, computed at the wavelength of the laboratory trapping device laser. From the calculations of the particle's cross-section, discussed in Section 3.4, we

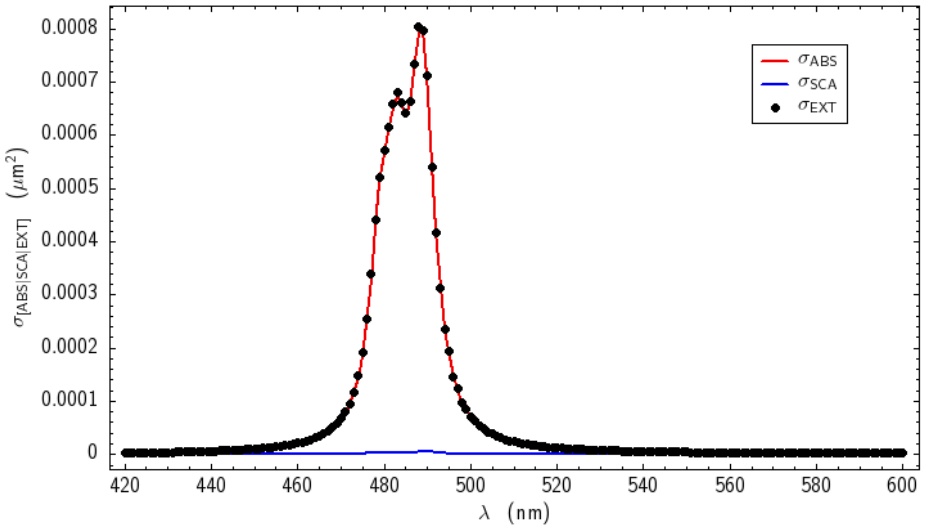

Figure 3.2: Absorption (red line), scattering (blue line), and extinction (black dots) cross-sections computed for the porphyrin nanotube of our test model as functions of wavelength. Extinction is the sum of absorption and scattering processes, in this case nearly completely dominated by the former over the latter.

have seen that the particle has an absorption dominated cross-section peak close to 494 nm. We can, therefore, try extracting the particle's T-matrix at a wavelength of 488 nm, characteristic of visible laser laboratory trapping devices. For this, we need to modify the input data files.

In our initial configuration (`DEDFB_24`), the wavelength of 488 nm is the 69^{th} computed wavelength. We therefore need another input file (which we call `DEDFB_488nm`), with the following contents:

```
24   0
 1.7755D00  3.0000000D08    1.000000D00   0    1  0  3
4.8800000e-07
    1    1    1    1    1    1    1    1    1    1    1    1    1    1 \
  1    1
    1    1    1    1    1    1    1    1
  1  5.00d-9
 1.000000D00
 ( -3.0538312819720566 , 2.511362127727128 )
```

```
0
```

This is a variation of `DEDFB_24`, requesting the calculation for only a single wavelength (488 nm) and declaring the corresponding optical function of the material. Similarly, we need a modified version of `DCLU_24`, which we call `DCLU_488nm`, with the following form:

```
  24    3    5  5760   1  149  300    0    0
   0.0000000e+00   1.0000000e-08   0.0000000e+00
   0.0000000e+00  -1.0000000e-08   0.0000000e+00
   8.5355339e-09   5.0000000e-09   0.0000000e+00
   8.5355339e-09  -5.0000000e-09   0.0000000e+00
  -8.5355339e-09   5.0000000e-09   0.0000000e+00
  -8.5355339e-09  -5.0000000e-09   0.0000000e+00
   0.0000000e+00   1.0000000e-08   1.0000000e-08
   0.0000000e+00  -1.0000000e-08   1.0000000e-08
   8.5355339e-09   5.0000000e-09   1.0000000e-08
   8.5355339e-09  -5.0000000e-09   1.0000000e-08
  -8.5355339e-09   5.0000000e-09   1.0000000e-08
  -8.5355339e-09  -5.0000000e-09   1.0000000e-08
   0.0000000e+00   1.0000000e-08   2.0000000e-08
   0.0000000e+00  -1.0000000e-08   2.0000000e-08
   8.5355339e-09   5.0000000e-09   2.0000000e-08
   8.5355339e-09  -5.0000000e-09   2.0000000e-08
  -8.5355339e-09   5.0000000e-09   2.0000000e-08
  -8.5355339e-09  -5.0000000e-09   2.0000000e-08
   0.0000000e+00   1.0000000e-08   3.0000000e-08
   0.0000000e+00  -1.0000000e-08   3.0000000e-08
   8.5355339e-09   5.0000000e-09   3.0000000e-08
   8.5355339e-09  -5.0000000e-09   3.0000000e-08
  -8.5355339e-09   5.0000000e-09   3.0000000e-08
  -8.5355339e-09  -5.0000000e-09   3.0000000e-08
 0.00D01  0.00D00  0.00D01  180.00D00  0.00D01  0.00D01
 0.00D01  0.00D00  0.00D01  0.00D00  0.00D01  0.00D01
 1
0
USE_REFINEMENT=1
USE_DYN_ORDERS=0
```

Again, this file is very similar to `DCLU_24`, except for declaring `JWTM` = 1 instead of '0' and for using two runtime options (iterative matrix inver-

sion refinement and dynamic orders; both will be discussed in detail in the next chapter). Both `DEDFB_488nm` and `DCLU_488nm` are included in the `src/test_data/trapping` folder and they can be simply copied to the working directory.

We can now create a new folder and run the T-matrix calculation:

```
WORK_DIR$ mkdir tra_results
WORK_DIR$ OMP_NUM_THREADS=1,NUM_WORK_THREADS [mpirun -n 1] \
  INSTALL_DIR/build/cluster/np_cluster DEDFB_488nm DCLU_488nm tra_results
```

where the `mpirun -n 1` is only needed if the code was compiled with *MPI* enabled. We can therefore descend in the `tra_results` folder and copy the input data for the trapping calculation here:

```
tra_results$ cp INSTALL_DIR/test_data/trapping/DFRFME_488nm .
tra_results$ cp INSTALL_DIR/test_data/trapping/DLFFFT .
```

We can now run the trapping calculation with:

```
tra_results$ INSTALL_DIR/build/trapping/np_trapping DFRFME_488nm DLFFFT .
```

The output of `np_trapping` consists in a set of binary files, containing calculation information, and in some plain text files, containing the radiation forces cross-sections (`c_force_cs.txt`), the radiation torque cross-sections (`c_torque_cs.txt`) and the grid scaling in meters (`c_grid_scale.txt`). The details on the contents of these files are presented in Section 2.7.2. The validation of the final results can be tested via numerical match, using the `pycompare.py` number parsing script to compare the final output with the precomputed case. The comparison of the grid scale can be run by:

```
tra_results$ INSTALL_DIR/src/scripts/pycompare.py --ffile \
  ~/Programming/gits/np_tmcode/test_data/trapping/c_grid_scale_488nm.txt \
  --cfile c_grid_scale.txt
INFO: using line-wise mode
INFO: counting result lines...
INFO: the output files have 21 lines
INFO: checking file contents... 100%
ERROR COUNT: 0
WARNING COUNT: 0
NOISE COUNT: 0
SUCCESS: c_grid_scale.txt is consistent with \
  /home/lamura/Programming/gits/np_tmcode/test_data/trapping/ \
```

```
  c_grid_scale_488nm.txt
```

Similarly, the test on the force component cross-sections is:

```
tra_results$ INSTALL_DIR/src/scripts/pycompare.py --ffile \
  ~/Programming/gits/np_tmcode/test_data/trapping/c_force_cs_488nm.txt \
  --cfile c_force_cs.txt
INFO: using line-wise mode
INFO: counting result lines...
INFO: the output files have 21 lines
INFO: checking file contents... 100%
ERROR COUNT: 0
WARNING COUNT: 0
NOISE COUNT: 0
SUCCESS: c_force_cs.txt is consistent with \
  /home/lamura/Programming/gits/np_tmcode/test_data/trapping/ \
  c_force_cs_488nm.txt
```

and the one on the torque component cross-sections is:

```
tra_results$ INSTALL_DIR/src/scripts/pycompare.py --ffile \
  ~/Programming/gits/np_tmcode/test_data/trapping/c_torque_cs_488nm.txt \
  --cfile c_torque_cs.txt
INFO: using line-wise mode
INFO: counting result lines...
INFO: the output files have 21 lines
INFO: checking file contents... 100%
ERROR COUNT: 0
WARNING COUNT: 0
NOISE COUNT: 0
SUCCESS: c_torque_cs.txt is consistent with \
  /home/lamura/Programming/gits/np_tmcode/test_data/trapping/ \
  c_torque_cs_488nm.txt
```

Chapter 4

Model creation and editing

4.1 Concepts for advanced modeling

`NP_TMcode` enables the user to obtain full T-matrix based solution of extinction and scattering processes involving particles of arbitrary shape in a much faster way than what was made possible by the legacy implementation running on the same hardware. The use of optimized calculation libraries alone, for a single computing core, reduces the calculation time by a factor of 8 with respect to the original code. In addition, the possibility to use multiple parallel processes and threads can introduce further speed-up factors, which, for proper hardware and model complexity, scales almost linearly with the ratio between the number of computed wavelengths over the number of computing cores (i. e. $T_{calc} \propto n_\lambda / n_{proc}$). However, even though modern hardware solutions offer the possibility to increase the complexity of models, the calculations obviously need to be planned with the available resource limits in mind. Therefore, while `NP_TMcode` can be executed on any architecture supporting its dependencies, from laptop computers and workstations up to exa-scale computing farms, the resources requested by the problem under investigation must match the limitations of the host hardware.

The calculation of scattering and extinction processes using the T-matrix implies that the particle is represented by a matrix of $[N \times N]$ complex elements, stored as pairs of double precision floating point values (i. e. 16

bytes per complex element). Since N is given by:

$$N = 2\, n_{sph}\, l_{max}\, (l_{max} + 2), \tag{4.1}$$

where n_{sph} is the number of spheres used to model the particle and l_{max} is the maximum field expansion order accounted by the calculation to achieve the required precision on a converging, but theoretically infinite, series of discrete contributions, the T-matrix itself can quickly become problematic to handle. The size of the T-matrix is determined by the ratio among the particle dimensions (or the features that need to be solved in it) and the radiation wavelength. Spherical components whose size is a considerable fraction of the radiation field wavelength (say $r_{part} > 0.1\,\lambda$) require increasingly large values of field expansion orders to achieve convergence, while small spherical units need to be used in large numbers, in order to properly fill a substantially bigger model particle. For particle radii $r_{part} \ll 0.1\,\lambda$ or $r_{part} \gg 10\,\lambda$ the full treatment is no longer strictly required and faster, approximate methods can be applied, thus effectively limiting the range where an exact formal solution is needed.

As an example, the particle shown in Fig 4.1, used to represent a simple core-mantle structure for a sub-μm scale astrophysical dust grain, if solved for an UV wavelength of $\lambda = 100\,\text{nm}$ requires a field expansion order of $l_{max} = 6$, which, by using Eq 4.1 for $n_{sph} = 42$, results in a $[4032 \times 4032]$ elements T-matrix, corresponding to 0.24 GiB of data. Such an example model can be easily treated even by commercial laptops. However, although the T-matrix is one of the largest data structures handled by the code, it only represents a fraction of the necessary data. Other structures, such as electromagnetic field descriptors, workspaces for algebraic operations, storage of results, and process synchronization data, also contribute to increase the memory requirements. Since the size of the necessary datasets is typically large and it depends on the nature of the problem, `NP_TMcode` relies on the use of the heap memory to dynamically manage the available resources. In addition, if `NP_TMcode` is run in its most optimized configuration, using external libraries and parallel computing, the total amount of required resources is subject to an overhead whose exact value is affected by implementation, runtime configuration and model characteristics.

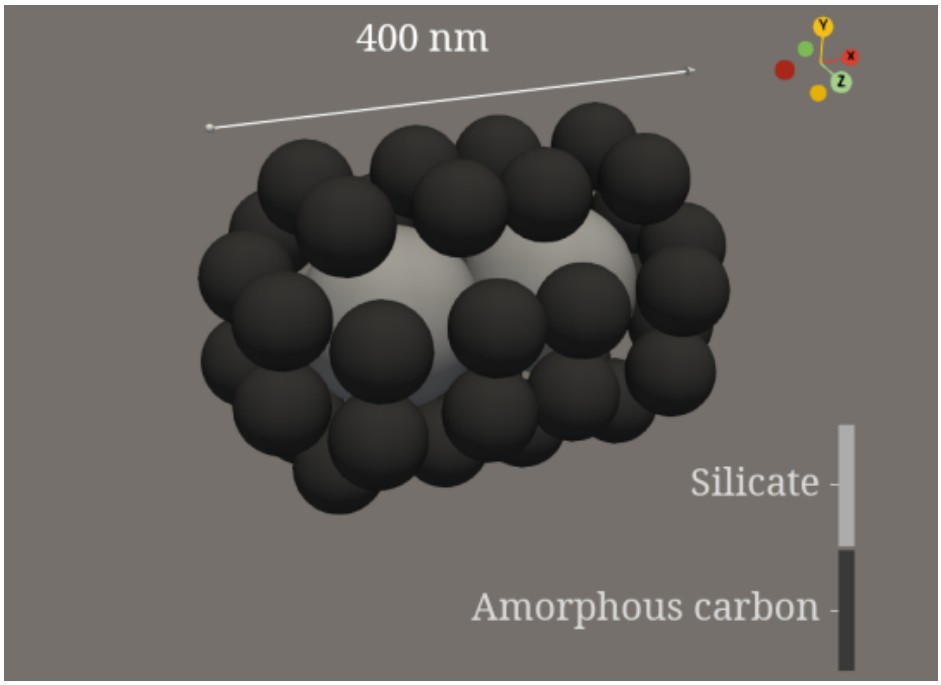

Figure 4.1: Example model to represent a sub-μm scale astrophysical dust grain, composed by a silicate core (modeled with 2 spheres with radius $r_{core} =$ 68.6 nm, shown in light gray) and an amorphous carbon mantle (formed by 40 spheres with radius $r_{mantle} = 34.3$ nm, shown in dark gray shading). The ruler shown above the particle represents the particles largest extension.

In order to assist the user in evaluating the system requirements for a specific calculation, `NP_TMcode` provides ways to estimate the memory consumption, both in the stage of model definition and at runtime, together with the possibility to set threshold limits, which prevent the calculation from attempting to consume more resources than the allotted ones. Knowledge of parameters such as the memory requirements for solving one iteration of a model is the starting point to assess the feasibility of the calculation with the available hardware, or to choose the optimal parallelization strategy, when accessing scalable resources. The sections presented in this chapter provide guidelines to build advanced particle models, to estimate their complexity, in terms of data structures, and to configure the calculation accordingly.

4.2 Resource management

`NP_TMcode v0.10.5` takes advantage of parallel architectures in two fundamental ways: by dividing independent calculations at different wavelengths among sets of minimally communicating threads and processes (leveraging the embarrassingly parallel nature of the calculation steps that depend on wavelength), and by distributing intensive algebraic operations among work-sharing threads (using GPU offload capabilities, if available). The critical aspect that affects the configuration of a model run is that the execution blocks which run simultaneously must not exceed the hardware limits.

Taking as example the ASUS ZenBook with an Intel Core i9-13900H CPU supporting 20 threads and a NVIDIA GeForce RTX 4060 Ada Lovelace GPU with 8 GiB VRAM, used for development the model particle illustrated in Fig. 4.1, for which the T-matrix takes 0.24 GiB of memory space, can be safely solved by `np_cluster`, compiled with *MPI*, *OpenMP*, and with the `MAGMA` library enabled, using all the available threads simultaneously. Before starting the calculation, the code produces the following log to standard output:

```
1 DEBUG: Proc-0 found 1 GPU device.
2 INFO: Process 0 initializes MAGMA.
3 INFO: making legacy configuration... done.
4 INFO: particle radius is 1.7265e-07 m.
5 INFO: iteration data requires 0.29737 GiB of RAM.
6 INFO: code execution needs 10.408 GiB of RAM.
7 INFO: Size of matrices to invert: 4032 x 4032 (0.24225 GiB).
8 INFO: using MAGMA calls.
```

Lines 1 and 2 concern the hardware detection, reporting how many GPU devices are found by each *MPI* process (in this case 1 GPU device seen by a single *MPI* process), and whether the connection with the assigned device via `MAGMA` routines was successful. Line 3 informs the user about the construction of the configuration data from the input parameters (which are internally converted in the legacy format, for consistency tests), while line 4 reports the minimum radius in meters that encompasses the whole particle (a basilar piece of information to evaluate whether the solution achieved convergence, as it will be discussed later on). Lines 5, 6, and 7, instead, summarize the

estimated memory occupation according to the following scheme:

- line 5: gives the size of the data structures associated with a single iteration (the T-matrix, the field descriptors and the storage of results);

- line 6: estimates the total amount of data used by all simultaneous threads of the current process (in this case, the iteration data size of 0.297 GiB multiplied by 20 simultaneous iteration threads in the current process and by an overhead factor, here approximately equal to 1.75);

- line 7: reports the dimensions of the T-matrix expressed as the product of the number of rows and columns.

The critical point that needs to be evaluated, when planning a model and programming a parallelization strategy, is that each iteration process (or thread) expects to have its own memory space in the heap, in order to handle the data and the associated overheads. While the size of the data structures can be computed in advance with respect to the actual execution of the calculations, the overheads are not precisely known, because they depend on which external libraries were used in the optimization and on how they were compiled and implemented on the target hardware. A general expectation is that higher degrees of optimization lead to larger overheads, but these should generally count for a factor of less than 2.

When using GPUs to perform offloaded operations, very similar considerations must be applied to the limitations introduced by the VRAM available on the GPU. In this case, however, the `NP_TMcode` does not need to load all the iteration data to the device, which can instead work using just the space needed to allocate one T-matrix for each simultaneous wavelength calculation, multiplied by an overhead factor of approximately 1.2 times, to allow for auxiliary workspace and data vectors. In the example illustrated above, this estimate grosses up to $20 \times 1.2 \times 0.24\,\mathrm{GiB} = 5.8\,\mathrm{GiB}$ thereby fitting in the device's 8 GiB VRAM.

Using the `model_maker.py` utility script, the user will be prompted with preliminary information concerning the hardware requirements needed to solve a specific model ahead of the calculation execution. In particular, the

script calculates the size of the T-matrix and of the iteration data structures, inferring how many parallel calculations can be run by applications such as `np_sphere`, `np_cluster` and `np_inclusion`. This information helps in planning the execution of parallel calculations, booth indicating how many parallel calculations can be executed (thus helping to request the proper resources) as well as by addressing the model design towards a more efficient representation of the structure. The methods to control the calculation preparation and execution for `np_sphere`, `np_cluster`, and `np_inclusion` are presented in the next sections. `np_trapping`, instead, has a rather different parallel implementation, more oriented to task work-sharing, rather than solving distinct independent calculations, and it is therefore treated in a slightly different way.

4.3 Runtime settings

4.3.1 Execution environment

The runtime behavior of `NP_TMcode` can be modified by means of environment variables, launcher applications, and runtime options. Depending on whether `NP_TMcode` was configured and built with *MPI* and *OpenMP*, using launchers such as `mpirun|mpiexec -n NUMBER_OF_PROCESSES` (or `srun -n NUMBER_OF_PROCESSES` in a `SLURM` environment) with one of `np_sphere`, `np_cluster`, or `np_inclusion` will create a group of processes that work simultaneously on a distributed subset of the full wavelength range of the problem. Conversely, defining the conventional environment variable `OMP_NUM_THREADS="LEVEL_1,LEVEL_2"` will create a thread hierarchy, used by each application process, to further subdivide the wavelength scales among the thread groups corresponding to `LEVEL_1`, subsequently defining `LEVEL_2` work-sharing threads in each group. The best strategy to organize the parallel hierarchy depends on the characteristics of the problem that is being solved. Rather single particle models, with few differential calculations in addition to the direction-averaged one, are more efficiently solved by maximizing the number of parallel calculations devoted to different wavelengths. Conversely, more complicated models or calculations that require the explicit

solution of many specific scattering directions perform better with a smaller number of simultaneous processes (which relax memory requirements), with more threads devoted to work-sharing (which speed up the solution of deeply nested loops).

Another environment variable, which affects the code execution is `OMP_TARGET_OFFLOAD`, which can be set to "`DISABLED`", if one wishes to avoid offloading to GPUs (e. g., while solving small test cases, particularly for debug purposes), or "`MANDATORY`", if, on the contrary, the plan is to force the use of a GPU even in case the calculation would not strictly require it. In most cases, setting this variable is not necessary, since the default behavior of using the GPU depending on convenience is a solid strategy.

Using *MPI* processes or *OpenMP* threads to split the wavelength range among parallel calculations on a single workstation without multiple GPUs leads to almost equivalent performance, since the inter-calculation communication steps for this parallel task have been reduced to minimal operations. However, if the calculation is run by a system that offers more than one GPU device, or it needs to be coordinated among different nodes of a computing farm, only *MPI* will allow for a proper distribution of the processes among the nodes and assign to each process a different GPU device, depending on the process rank. In any case, when computing the resource requirements, attention must be paid to the fact that the estimated runtime requirements are given *per process*. This appears to be a natural choice, because the user may be interested in calibrating the request of resources and the number of simultaneous calculations to be executed, depending on the available hardware.

4.3.2 Runtime options

In order to grant that the calculation fits within the assigned resources and achieves the highest possible accuracy, `NP_TMcode` accepts a set of runtime options that can be added to the input configuration files. These options can be added after the '0' end-of-data character code in the second configuration file of `np_sphere`, `np_cluster`, and `np_inclusion`, in addition to a separate set of options that can appear in both configuration files of `np_trapping`.

Use of runtime options has no effects on the original *FORTRAN* applications, because the corresponding input lines (located after the end of legacy input data) are not ingested. `NP_TMcode`, instead, parses all the contents of the input files. Lines starting with a hash character (‘`#`’) are treated as comments and ignored. All other lines must correspond to one of the recognized runtime options, otherwise the code execution terminates with an error message.

The allowed options for `np_sphere`, `np_cluster`, and `np_inclusion` must be given in the second configuration file and they are:

- `USE_DYN_ORDERS=1|0`: enable (‘1’) or disable (‘0’) the code ability to dynamically lower the maximum field multipolar expansion orders if the particle is too small for the current wavelength. This option is enabled by default because too high orders for a particle that is small with respect to the current wavelength can introduce numerical instability in the results.

- `USE_REFINEMENT=0|1`: enable (‘1’) or disable (‘0’) the iterative refinement to reduce numerical instability after matrix inversion. This option is not enabled by default, because implies the use of 3 times more memory than the T-matrix size in the process of matrix inversion (therefore increasing the amount of required VRAM by the same factor, when using a GPU).

- `HOST_RAM_GB=ram_gb`: maximum amount of memory in GiB that a single `NP_TMcode` *process* is allowed to request from the host. `NP_TMcode v0.10.5` is designed to terminate with an error code if a process breaks this limit. The limit needs not to be the whole host memory (e.g., a safe run of the code with 2 *MPI* processes on a machine that features 200 GiB of run would set this limit to 90 GiB, so that the whole calculation would issue an error message and stop, if more than 180 GiB are necessary, instead of attempting to get the resources). This setting is optional. Omitting it, or setting it to ‘0’ disables the test on resource requirements and performs a direct calculation attempt. A safety limit can be specified as a positive number (integer or floating point, with or without ’`e`’-formatted scientific notation).

- `GPU_RAM_GB=vram_gb`: same as the previous, but for the VRAM of a GPU device. Again, it must be recalled that the opted setting is for a single *MPI* process. It is generally advisable that a machine runs as many *MPI* processes as the number of available GPUs (if they need to be used). Multiple *MPI* processes can work with the same GPU, but, in this case, they should not be assigned more than the corresponding fraction of available VRAM.

`np_trapping`, on the other hand, accepts the following option:

- `PRECOMPUTED=preset_name`: the name of a pre-computed focal field description (usually a binary file named '`c_TFRFME.hd5`'). If run with this option, `np_trapping` will skip the calculation of the radiation field grid (which is the most computationally intensive task of the application). By setting this option, a single focal field descriptor can be used with several particle models, without repeating the calculation of the radiation field properties. If present, this option **must be** specified in the *first* configuration file.

The recommended way to configure the calculation is to use dynamic orders and, if the hardware resources allow it, iterative refinement. The reason behind this strategy is connected with the discussion presented in Section 4.1 concerning the properties of the T-matrix and their relation with hardware requirements. In essence, the T-matrix is obtained by the inversion of a field continuity matrix, which is composed by the sums over the field expansion in spherical vector harmonic components (Borghese et al., 2007). These vector harmonics span over an increasing range of orders of magnitude. As the order of the calculation increases, the fixed precision of the numeric representation of the matrix element may lead to loss of accuracy on the smallest elements, with respect to the largest ones. When the matrix is inverted, this loss of accuracy leads to errors on the largest elements, i.e. the matrix inversion becomes numerically unstable.

`NP_TMcode` addresses the problem of matrix inversion numerical stability in two ways:

- by dynamically assigning the calculation order for each wavelength;

- by using iterative matrix inversion refinement to improve accuracy.

The dynamic assignment of calculation orders is based on the fact that the size of the particle must be compared with the wavelength of the radiation field being currently evaluated. Particles that need to be solved at high orders for short wavelengths can be also solved at lower orders, when working at longer wavelengths. Therefore, the default behavior of `NP_TMcode` is to lower the order of calculation, if it is too high for the current wavelength, thus preventing instability effects. Increasing the order beyond the maximum value set by the user, on the contrary, is not supported, due to the danger of breaking hardware resource limits.

The iterative refinement process, instead, works on the fact that, provided that the first matrix inversion $\hat{\mathbf{A}}_k$ is a reasonable numeric approximation of the real inverse matrix $\mathbf{A}^{-1}$, we can define a residual matrix $\mathbf{R}_k$:

$$\mathbf{R}_k = I - \mathbf{A} \cdot \hat{\mathbf{A}}_k, \tag{4.2}$$

where I is the identity matrix, such that the new matrix:

$$\hat{\mathbf{A}}_{k+1} = \hat{\mathbf{A}}_k \cdot \mathbf{R}_k + \hat{\mathbf{A}}_k \tag{4.3}$$

will be a more accurate approximation of the inverse. Choosing one of the many available numeric implementations of complex matrix inversion to obtain a first numeric approximation of the inverted matrix in Eq. (**??**), `NP_TMcode` applications can iterate Eqs. (4.2) and (4.3) on k until either $\mathbf{R}$ contains only elements whose absolute value lies below a predefined threshold, or a maximum number of refinement iterations was reached (in which case a warning message is issued). This strategy reduces the chances of numeric instability at high orders, but it is not enabled by default, due to its heavier hardware resource requirements.

4.4 Building models

In the T-matrix formalism, particles of arbitrary shapes are represented as aggregates of spherical components, usually referred to as monomers, which

are arranged in space in a way that approximates the structures to be modeled. The monomers can be identical or different in size and composition and they can have spherically symmetric concentric layers of different materials. As discussed in Chapter 2, `NP_TMcode` uses a set of input parameters to control the distribution and the composition of the spheres that compose a model. In order to execute an extinction calculation, the input must describe:

- the incident radiation field (in terms of polarization and wavelength range);
- the properties of the different types of spheres used to model the particle (namely, the sphere sizes, the optical properties of their materials and their layer structures);
- the spatial distribution of the spheres;
- the maximum order of the calculations and any specific scattering geometry that should be solved together with the directionally average integrated solution.

These parameters are read from formatted configuration files, which can be used both by the original *FORTRAN* code implementation and by the new `NP_TMcode` applications. Since the structure of these files, presented in Chapter 2, is rigidly oriented to the parsing carried out by the original applications, which did not support labels, comment sections, or custom sorting of the input, `NP_TMcode` features a *Python* script named `model_maker.py`, located under the `src/scripts` folder, whose aim is to simplify the process of building model particles.

`model_maker.py` can produce particle input models for `np_sphere`, `np_cluster`, and `np_inclusion`, while `np_trapping` does not need a particle input model, because its description of the particle is embedded in the T-matrix file that is produced by one of the other three applications for the wavelength of interest. The script accepts the definition of the model input parameters in `YAML` format, which introduces a more explicit labeling of the relevant parameters, a hierarchical organization of the parameters depending on their role, and some degree of freedom in the order of parameter definition

(though there are reciprocally related parameters, such as, for instance, the wavelengths of the radiation field and the corresponding optical properties of the materials, which must be sorted according to the same logic, in order to be properly interpreted). The scripts supports different operating modes, which give various degree of control on the final particle structure. These range from a completely customized particle structure (useful for modeling a predetermined shape) to random particle generation with a set of monomer types.

The `NP_TMcode` software package includes various examples of model configuration files, under the `test_data` folder structure. The model maker script can be used to obtain the input configuration files from a `YAML` description file by issuing:

```
WORK_DIR$ INSTALL_DIR/src/scripts/model_maker.py CONFIG_FILE
```

where `CONFIG_FILE` is the full path to a valid configuration `YAML` file. In order to be a valid input for `NP_TMcode`, the configuration file needs to include the following sections:

- `system_settings`: a description of the system that will run the calculation (not necessarily the one building the model);
- `input_settings`: some options that control how the `NP_TMcode` code input files are written;
- `particle_settings`: settings that control the components of the particle model;
- `material_settings`: settings concerning the particle materials and the external medium;
- `radiation_settings`: settings that describe the incident radiation field;
- `geometry_settings`: settings to define the structure of the particle and the geometry of any possible specific differential calculation to be added.

In case runtime options are needed to modify the default behavior of the code, they can be included with an optional `runtime` section. The example configuration files distributed with the code package contain settings to create

some of the pre-computed models, with comment lines describing the role of each parameter. In what follows, we present the definition of the input model configuration files for the cases of fully customized particle models and that of random particle generation.

4.4.1 User defined particle structure

The strategy to obtain a totally controlled particle structure is simply by giving an explicit definition of all the model configuration parameters, including the description of the monomers and their distribution in space. A valid `YAML` description file for a custom particle would contain the following sections:

```
1 system_settings:
2   # Limit on host RAM use in Gb (0 for no configuration limit)
3   max_host_ram : 0
4   # Limit on GPU RAM use in Gb ( 0 for no configuration limit)
5   max_gpu_ram  : 0
6   # OBJ model export flag (requires pyvista; 0 is FALSE)
7   make_3D      : 1
```

This is the system requirements section, where the limits on hardware resources have been set to '0', because we are not interested in setting safety parameters for the case of a small test particle. We shall illustrate the use of some limits with the random particle generator, which is more likely to be used for models of larger particles. The optional argument '`make_3D : 1`' in line 7 tells the script ti attempt using the *pyvista* module to build a 3D representation of the particle in `OBJ` format (a formatted plain text file containing information to render 3D objects, which is used by most 3D drawing applications).

```
8 input_settings:
9   # Folder to write the code input configuration files
10  input_folder : "test_subdir"
11  # Name of the scatterer description file
12  spheres_file : "DEDFB"
13  # Name of the geometry description file
14  geometry_file: "DCLU"
```

The code above illustrates the `input_settings` section, which specifies where the input configuration files for the calculation will be written (line 10, where `input_folder` needs to point to an existing folder) and what will be the names of the first and the second configuration files (lines 12 and 14, respectively).

```
15 output_settings:
16   # Index of the scale for transition matrix output
17   jwtm         : 1
```

This is a short section, identifying which wavelength scale (if any) will be the one for which the T-matrix shall be written to an output file for subsequent use. In this case the choice is '1', indicating that the first wavelength will save the T-matrix.

```
18 particle_settings:
19   # What application to use (SPHERE | CLUSTER | INCLUSION)
20   application : "CLUSTER"
21   # Number of spheres
22   n_spheres   : 4
23   # Number of sphere types
24   n_types     : 4
25   # Vector of sphere type identifiers (what type is each sphere)
26   sph_types   : [ 1, 2, 3, 4 ]
27   # Vector of layers in types (how many layers in each type)
28   n_layers    : [ 1, 1, 1, 1 ]
29   # Spherical monomer radii in m (one size for each type)
30   radii       : [ 5.0e-8, 4.0e-8, 7.5e-8, 3.0e-8 ]
31   # Layer fractional radii (one per layer in each type)
32   rad_frac    : [ [ 1.0 ], [ 1.0 ], [ 1.0 ], [ 1.0 ] ]
33   # Index of the dielectric functions (one per odd layer in each type)
34   #
35   # 1 is first file in 'dielec_file', 2 is second ...
36   dielec_id   : [ [ 1 ], [ 1 ], [ 1 ], [ 1 ] ]
```

This section describes what application will be used to solve the particle model (line 20), how many spheres will compose the particle (line 22), how many types of spheres will be used (line 24). In this example we are using 4 spheres of 4 different types, i. e. one sphere per type. The following lines control what type each sphere belongs to (line 26, how many concentric layers are in each type (line 28), the radii of each type of sphere (line 30, in units of

m), the fraction of the sphere radius which reached by the outer shell of the layer (1.0 is a layer reaching the outer surface of the sphere), and a positional index, corresponding to the index of the dielectric properties to be used for the material in the layer (line 36, further detailed in the `material_settings` section).

Before describing the other sections, it might be useful to show some of the possible variations on these settings. If, for example we were interested in using two different materials in sphere type 1, setting the transition at 0.75 of the sphere radius, we would use the following parameters (recall that we need a transition layer, so we need 3 layers):

```
# Vector of sphere type identifiers (what type is each sphere)
sph_types   : [ 1, 2, 3, 4 ]
# Vector of layers in types (how many layers in each type)
n_layers    : [ 3, 1, 1, 1 ]
# Spherical monomer radii in m (one size for each type)
radii       : [ 5.0e-8, 4.0e-8, 7.5e-8, 3.0e-8 ]
# Layer fractional radii (one per layer in each type)
rad_frac    : [ [ 0.749, 0.75, 1.0 ], [ 1.0 ], [ 1.0 ], [ 1.0 ] ]
# Index of the dielectric functions (one per odd layer in each type)
#
# 1 is first file in ‘dielec_file‘, 2 is second ...
dielec_id   : [ [ 2, 1 ], [ 1 ], [ 1 ], [ 1 ] ]
```

If, on the contrary, we were interested in using 4 spheres of only two types, using multiple layers for the first type and a single layer for the second, we would set:

```
# Number of spheres
n_spheres   : 4
# Number of sphere types
n_types     : 2
# Vector of sphere type identifiers (what type is each sphere)
sph_types   : [ 1, 1, 2, 2 ]
# Vector of layers in types (how many layers in each type)
n_layers    : [ 3, 1 ]
# Spherical monomer radii in m (one size for each type)
radii       : [ 5.0e-8, 4.0e-8 ]
# Layer fractional radii (one per layer in each type)
rad_frac    : [ [ 0.749, 0.75, 1.0 ], [ 1.0 ] ]
```

```
  # Index of the dielectric functions (one per odd layer in each type)
  #
  # 1 is first file in ‘dielec_file‘, 2 is second ...
  dielec_id   : [ [ 2, 1 ], [ 1 ] ]
```

Moving onto the `material_settings` section, we find:

```
37 material_settings:
38   diel_flag   : 0
39   # External medium dielectric constant
40   extern_diel : 1.00e0
41   # Dielectric function files folder
42   dielec_path : "."
43   # List of dielectric function files (used if diel_flag = 0)
44   dielec_file : [ "eps_ashok_C.csv" ]
45   # Matching method between optical functions and radiation wavelengths
46   #
47   # INTERPOLATE: the functions are interpolated on wavelengths
48   # GRID: only the wavelengths with defined functions are computed
49   #
50   match_mode  : "INTERPOLATE"
51   # Reference dielectric functions (used if diel_flag = -1)
52   #
53   # One real and one imaginary part for each odd layer in each type.
54   diel_func   : [ ]
```

These are the settings of the materials used to model the particle. The `diel_flag` parameter of line 38 corresponds to the `IDFC` flag to describe the material dielectric functions (introduced in Section 2.4.1). Similarly, the `extern_diel` of line 40 is the external medium dielectric constant at the reference wavelength, corresponding to `EXDC`. The `dielec_path` of line 42 is the file system location where the script will look for the material dielectric function table files (setting it to "`.`" implies that the data files will be sought for in the current working directory). Some tabulated dielectric function files are distributed with the software under the "`ref_data`" folder. The `dielec_file` parameter of line 44 is a list of dielectric function table files. With the `dielec_id` parameter set as in line 36, only one material is used. However, a setting like:

```
  dielec_id   : [ [ 2, 1 ], [ 1 ] ]
```

would require the definition of two materials, such as, for example:

```
dielec_file : [ "eps_ashok_C.csv", "eps_draine_Si.csv" ]
```

using the second file (astronomical silicates, according to Draine and Flatau, 1994) as the innermost layer of sphere type 1 and the first one (amorphous carbon, according to Palik, 1991) for its second layer and for the only layer of sphere type 2. The `match_mode` parameter of line 50 controls how the dielectric functions (usually given in terms of their dependence on wavelength) should be related with the calculation wavelength range. This parameter can be set either to '`INTERPOLATE`', which performs an interpolation of the dielectric function on the wavelength range considered by the code, or to '`GRID`', which, instead, forces the code to run only on the wavelengths that are explicitly defined in the tabulated dielectric functions. The last parameter of the section, i. e. `diel_func` in line 54 can be used to explicitly define the dielectric functions of each non-transition (odd) layer in each sphere type at the reference wavelength, by explicitly writing them as pairs of real and imaginary parts, if the `diel_flag` parameter is set to a value smaller than 0. Following on, the properties of the radiation field are controlled by the section:

```
55 radiation_settings:
56   # Radiation field polarization (LINEAR | CIRCULAR)
57   polarization: "LINEAR"
58   # First scale to be used
59   scale_start : 4.0e-07
60   # Last scale to be used
61   scale_end   : 5.0e-07
62   # Calculation step (overridden if 'match_mode' is GRID)
63   scale_step  : 5.0e-08
64   # Peak Omega
65   wp          : 3.000e15
66   # Peak scale
67   xip         : 1.000e00
68   # Define scale explicitly (0) or in equal steps (1)
69   step_flag   : 0
70   # Type of scaling variable
71   scale_name  : "WAVELENGTH"
```

This section starts with the radiation field `polarization` parameter (line

57), which can be set to 'LINEAR' or 'CIRCULAR', followed by the definition of the spectral range covered by the calculation. The spectral range can in principle be defined in terms of wavelength, pulsation, energy, or particle size parameter. Since the dielectric functions used in this example are expressed in wavelength, the spectral range follows the same convention defining a scale_start of 400 nm (line 59, a scale_end of 500 nm (line 61, and an incremental step of 50 nm (line 63). This last setting would be overridden by the dielectric function table grid, if match_mode is set to 'GRID'. The parameter wp on line 65 is the pulsation of the reference wavelength, whose dimension woudl correspond to a scale of 1, while the parameter xip is used to normalize the size scale if the size parameter is used as the scaling variable, by setting scale_name : "XI" in line 71. The step_flag parameter of line 69 corresponds to the INSTPC parameter of Section 2.4.1, and it switches from the explicit definition of an arbitrary sequence of input wavelengths (not necessarily regularly spaced), obtained by setting '0' as value, or a regular grid of wavelengths, separated by the same interval.

The geometry of the problem and the definition of the integration intervals are defined in the geometry section, which takes the following form:

```
72 geometry_settings:
73   # Maximum internal field expansion
74   li          : 8
75   # Maximum external field expansion (not used by SPHERE)
76   le          : 8
77   # Number of transition layer integration points
78   npnt        : 149
79   # Number of non transition layer integration points
80   npntts      : 300
81   # Averaging mode
82   iavm        : 0
83   # Meridional plane flag
84   isam        : 0
85   # Starting incidence azimuth angle
86   in_th_start : 79.0
87   # Incidence azimuth angle incremental step
88   in_th_step  : 10.0
89   # Ending incidence azimuth angle
90   in_th_end   : 89.0
```

```
# Starting incidence elevation angle
in_ph_start : 0.0
# Incidence elevation angle incremental step
in_ph_step  : 10.0
# Ending incidence elevation angle
in_ph_end   : 10.0
# Starting scattered azimuth angle
sc_th_start : 34.0
# Scattered azimuth angle incremental step
sc_th_step  : 15.0
# Ending scattered azimuth angle
sc_th_end   : 49.0
# Starting scattered elevation angle
sc_ph_start : 5.0
# Scattered elevation angle incremental step
sc_ph_step  : 5.0
# Ending scattered elevation angle
sc_ph_end   : 10.0
# Vector of sphere X coordinates (one per sphere or empty for random)
x_coords    : [
  0.00e00,
  0.00e00,
  1.18882e-07,
  -5.656855e-08,
  ]
# Vector of sphere Y coordinates (one per sphere or empty for random)
y_coords    : [
  3.00e-08,
  3.00e-08,
  3.00e-08,
  -2.656855e-08
  ]
# Vector of sphere Z coordinates (one per sphere or empty for random)
z_coords    : [
  0.00e00,
  9.00e-08,
  3.8627e-08,
  0.00e00
  ]
```

These parameters follow nearly the same convention of the geometry configu-

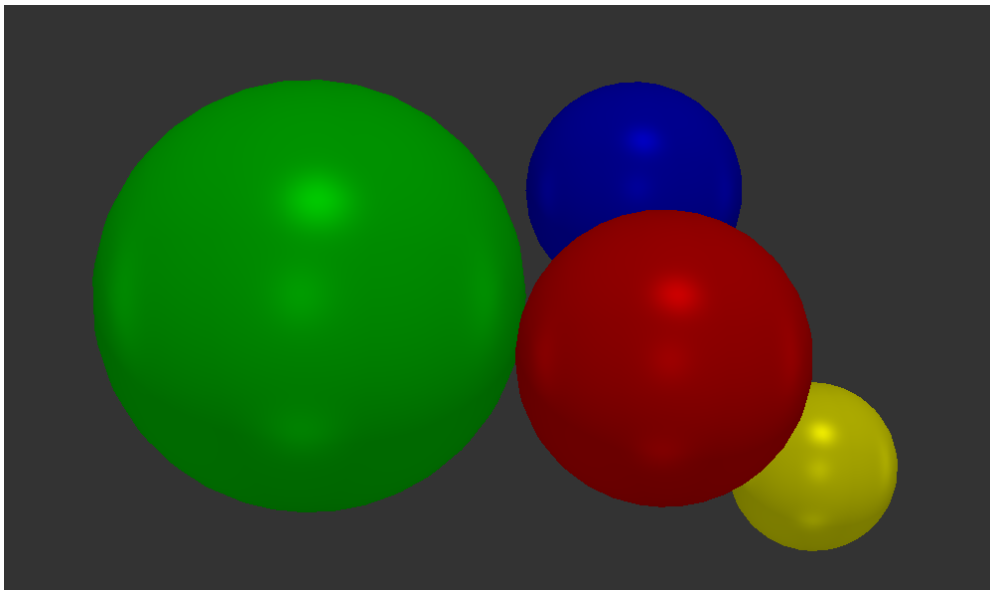

Figure 4.2: 3D representation of the particle corresponding to the custom model described in Section 4.4.1. Here the different colors are used to show spheres of different types.

ration parameters presented in Sections 2.4.1, 2.5.1, and 2.6.1, with `li` being the maximum internal field expansion order (the only one used by `np_sphere`, `le` the external, `npnt` and `npntts` the integration steps of non-transition and transition layers, `iavm` and `isam` the flags to control the output averaging mode and the reference plane for the geometry. The code section between lines 86 and 108 defines the sequence of differential radiation field incidence and scattering directions that need to be explicitly accounted for, while the code lines between 109 and 129 define the x, y, and z coordinate vectors of the centers of all the spheres.

The sections of the `YAML` configuration file, as well as their child parameters, may appear in any order, but the presence of all the parameters presented in this section, except `make_3D`, is necessary, in order for the script to properly handle the model. Provided that the configuration was defined as in the discussed example, and that the script is executed on a system featuring the *pyvista* module, the model particle should have the structure shown in Fig. 4.2.

4.4.2 Random particle generation

The random particle generation uses almost the same input as the custom model definition described in the previous section, with the exception of a few parameters to control the random engine. Specifically, these parameters are:

- `rnd_seed`: an integer number that acts as a seed for the sequence of pseudo-random values extracted from the system random generator (to be placed under the `system_settings` section);

- `rnd_engine`: one of '`COMPACT`' or '`LOOSE`', to produce a highly packed lattice structure or a more open irregular one (to be placed under the `system_settings` section);

- `max_rad`: the maximum allowed encircling radius for the model particle (expressed in m, by a floating point or by an '`e`'-formatted scientific notation number to be placed under the `particle_settings` section).

By defining the above optional parameters, the script enables random feature generation. In `NP_TMcode v0.10.5`, the `model_maker.py` script can work with random sphere type assignment and/or random sphere positions. These can be activated by setting the corresponding parameters to empty vectors. At present, no support is provided for the random generation of a sphere type, though this option may be planned for the future. The choice of random sphere types is activated by setting:

```
particle_settings:
  ...
  sph_types   : [ ]
```

while random sphere positions are obtained by setting:

```
geometry_settings:
  ...
  x_coords    : [ ]
  y_coords    : [ ]
  z_coords    : [ ]
```

The `LOOSE` particle generator starts from a single sphere at the center of the coordinates, then it attaches additional spheres using a growth scheme,

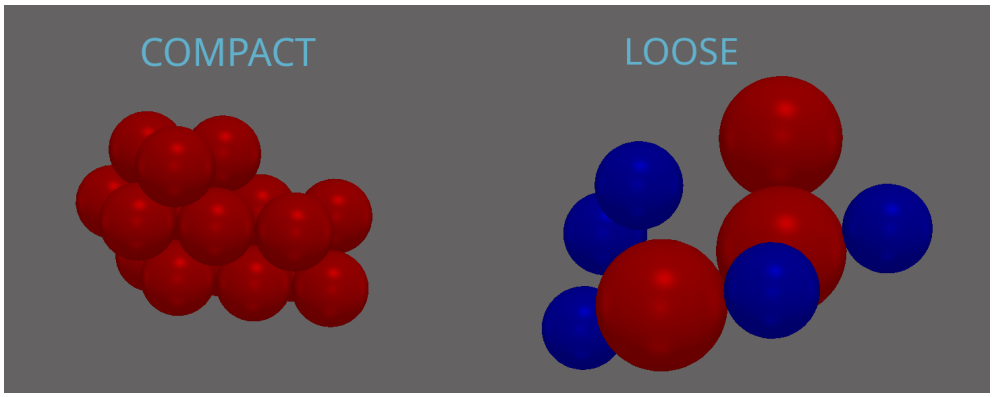

Figure 4.3: Examples of random generated particle models: a particle formed by identical monomers using the `COMPACT` engine (left) and a particle formed by two types of monomers, with different radii, built by the `LOOSE` engine (right, with blue and red spheres corresponding to the sphere types).

where new spheres start at the center and then move outwards until they become tangent to the outermost point reached by the previous ones in that direction. The process iterates until the desired number of spheres has been reached, or the volume contained within `rad_max` cannot accommodate further spheres. In the latter case, the script issues a warning about not being able to use all the requested spheres. The `COMPACT` particle generator, on the contrary, fills the allowed volume with spherical monomers distributed in a maximal filling factor packaging, then it subtracts the outermost sphere from a sequence of random directions, sculpting the model until it has the desired number of monomers. Because of its approach, the `COMPACT` generator has two major caveats:

1. it can only function if all the sphere types have the same radius (otherwise the maximal packaging calculation fails);

2. the radius of the monomers cannot be much smaller than the maximum allowable radius, otherwise a **huge number** of monomers will be generated and subsequently cherry-picked one by one, to sculpt the model, with possibly very long execution times.

Examples of model particles built with the `COMPACT` and the `LOOSE` parti-

cle generators are shown in Fig. 4.3. In the next section, we discuss the full configuration of a random particle model, showing how to work with configuration parameters to affect the final particle properties.

4.4.3 Editing the particle properties

Let's consider a case in which we want to model a $> 1\,\mu$m scale particle, using several smaller monomers (each with size of the order of $1\,\mu$m or below). We can take, as an example, the model of a condensed forsterite particle in an exo-planetary atmosphere. For illustration purposes, we can model this particle as an aggregate of 75 monomers, divided in two groups: large monomers ($r_{sph,1} = 1.259\,\mu$m) and small monomers ($r_{sph,2} = 0.794\,\mu$m). Such particle can comfortably fit within a radius of $r_p = 10.0\,\mu$m. Since we are using sphere types with different radii, and we are not planning to define the position of every single monomer, we can use the `LOOSE` particle random generator. A full configuration for this model would be:

```
system_settings:
  # Limit on host RAM use in Gb (0 for no configuration limit)
  max_host_ram : 200
  # Limit on GPU RAM use in Gb (0 for no configuration limit)
  max_gpu_ram  : 48
  # Random seed (a positive integer number)
  rnd_seed     : 15
  # Random engine (COMPACT or LOOSE)
  rnd_engine   : "LOOSE"
  # OBJ model export flag (requires pyvista; 0 is FALSE)
  make_3D      : 1

runtime:
  # Dynamically adapt orders
  dyn_orders   : 1
  # Use matrix inversion refinement
  refinement   : 1

input_settings:
  # Folder to write the code input configuration files
  input_folder : "."
  # Name of the scatterer description file
```

```
  spheres_file : "DEDFB_rnd_loose"
  # Name of the geometry description file
  geometry_file: "DCLU_rnd_loose"

output_settings:
  # Index of the scale for transition matrix output
  jwtm         : 1

particle_settings:
  # What application to use (SPHERE | CLUSTER | INCLUSION)
  application : "CLUSTER"
  # Number of spheres
  n_spheres   : 75
  # Number of sphere types
  n_types     : 2
  # Vector of sphere type identifiers (what type is each sphere)
  sph_types   : [ ] # empty means assign randomly
  # Vector of layers in types (how many layers in each type)
  n_layers    : [ 1, 1 ]
  # Spherical monomer radii in m (one size for each type)
  radii       : [ 1.25893e-6, 7.94328e-07 ]
  # Layer fractional radii (one per layer in each type)
  rad_frac    : [ [ 1.0 ], [ 1.0 ] ]
  # Index of the dielectric constants (one per layer in each type)
  #
  # 1 is first file in ‘dielec_file‘, 2 is second ...
  dielec_id   : [ [ 1 ], [ 1 ] ]
  # Maximum radius for random particle size in m
  max_rad     : 1.0e-5

material_settings:
  diel_flag   : 0
  # External medium dielectric constant
  extern_diel : 1.000e+0
  # Dielectric constant files folder
  dielec_path : "."
  # List of dielectric constant files (used if diel_flag = 0)
  dielec_file : [ "eps_forsterite.csv" ]
  # Dielectric constant files format (same for all files)
  dielec_fmt  : [ "CSV" ]
  # Matching method between optical constants and radiation wavelengths
```

```
  #
  # INTERPOLATE: the constants are interpolated on wavelengths
  # GRID: only the wavelengths with defined constants are computed
  #
  match_mode  : "GRID"
  # Reference dielectric constants (used if diel_flag = -1)
  #
  # One real and one imaginary part for each layer in each type.
  diel_const  : [ ]

radiation_settings:
  # Radiation field polarization (LINEAR | CIRCULAR)
  polarization: "LINEAR"
  # First scale to be used
  scale_start : 1.000e-6
  # Last scale to be used
  scale_end   : 2.500e-5
  # Calculation step (overridden if 'match_mode' is GRID)
  scale_step  : 1.0e00
  # Peak Omega
  wp          : 3.000e+15
  # Peak scale
  xip         : 1.000e00
  # Define scale explicitly (0) or in equal steps (1)
  step_flag   : 0
  # Type of scaling variable
  scale_name  : "WAVELENGTH"

geometry_settings:
  # Maximum internal field expansion
  li          : 10
  # Maximum external field expansion (not used by SPHERE)
  le          : 42
  # Number of transition layer integration points
  npnt        : 149
  # Number of non transition layer integration points
  npntts      : 300
  # Averaging mode
  iavm        : 0
  # Meridional plane flag
  isam        : 0
```

```
105  # Starting incidence azimuth angle
106  in_th_start : 0.0
107  # Incidence azimuth angle incremental step
108  in_th_step  : 0.0
109  # Ending incidence azimuth angle
110  in_th_end   : 0.0
111  # Starting incidence elevation angle
112  in_ph_start : 0.0
113  # Incidence elevation angle incremental step
114  in_ph_step  : 0.0
115  # Ending incidence elevation angle
116  in_ph_end   : 0.0
117  # Starting scattered azimuth angle
118  sc_th_start : 0.0
119  # Scattered azimuth angle incremental step
120  sc_th_step  : 0.0
121  # Ending scattered azimuth angle
122  sc_th_end   : 0.0
123  # Starting scattered elevation angle
124  sc_ph_start : 0.0
125  # Scattered elevation angle incremental step
126  sc_ph_step  : 0.0
127  # Ending scattered elevation angle
128  sc_ph_end   : 0.0
129  # Vector of sphere X coordinates (one per sphere or empty for random)
130  x_coords    : [ ] # empty means assign randomly
131  # Vector of sphere Y coordinates (one per sphere or empty for random)
132  y_coords    : [ ] # empty means assign randomly
133  # Vector of sphere Z coordinates (one per sphere or empty for random)
134  z_coords    : [ ] # empty means assign randomly
```

where we chose to use '15' as seed for our pseudo-random number sequence. For now, let's not consider the radiation field settings and just focus on the particle. If we run the `model_maker.py` script from a working directory that provides the `eps_forsterite.csv` data file (a copy of which is included in the `NP_TMcode` package under `ref_data`), like:

```
WORK_DIR$ INSTALL_DIR/src/scripts/model_maker.py CONFIG_FILE
```

we will get the input parameters for `np_cluster` in two files, named `DEDFB_rnd_loose` and `DCLU_rnd_loose`. The appearance of the particle should look

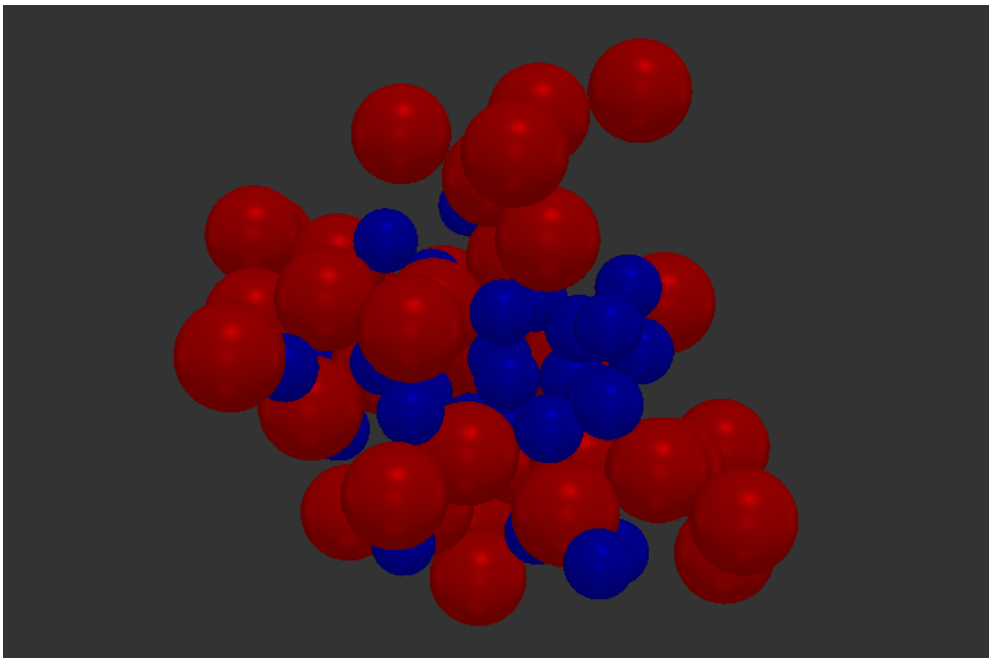

Figure 4.4: 3D representation of a random model particle obtained with a random combination of large monomers (red) and small monomers (blue), as detailed in Sec. 4.4.3. The maximum radius allowed for this particle is $r_p = 10.0\,\mu$m.

like Fig. 4.4. In addition, the script will write the following log to standard output:

```
INFO: estimated matrix size is 4.83 Gb.
INFO: system supports up to 9 simultaneous processes on GPU.
INFO: only 3 GPU processes allowed, if using refinement.
INFO: model requires 6.5614GiB of host RAM.
INFO: system supports up to 30 simultaneous processes
      (N.B.: not including overheads!)
INFO: the number of detected CPUs is 20.
INFO: smallest monomer radius is Rmin = 7.94328e-07m
INFO: largest monomer radius is Rmax = 1.25893e-06m
INFO: equivalent volume radius is Reqv = 4.60027e-06m
INFO: minimum encircling radius is Rcirc = 8.36871e-06m
```

The above information will play an important role at the time of calculation planning and execution.

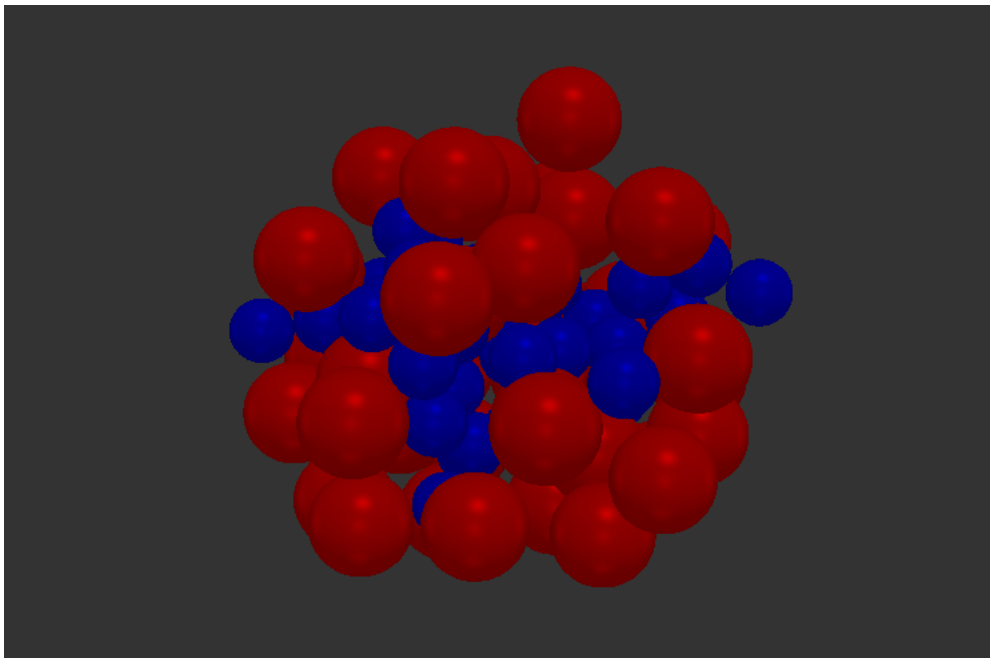

Figure 4.5: Same as Fig. 4.4, but replacing the maximum allowed radius of $r_p = 10.0\,\mu$m with a value of $7.1\,\mu$m.

A first look at the properties of the particle in Fig. 4.4 shows an irregular, fluffy structure. This could be fine for light particles built in low density environments, but it is not quite like the expectation for a particle that grows by condensation and aggregation. The structure of a particle can be changed in many ways. The first and simplest choice is to test different random seeds in line 7. However, changing the random seed does not affect the overall characteristics of the model, in terms of symmetry and fluffiness. This is a handy quick option, if the goal of the calculation is to work with a family of differently shaped particles with similar structural properties. A more sensible effect, instead, can be obtained by changing the maximum radius of the particle, to make it more compact. Modifying the `max_rad` parameter of line 51 to something like:

```
# Maximum radius for random particle size in m
max_rad     : 7.1e-6
```

and running the script with:

```
WORK_DIR$ INSTALL_DIR/src/scripts/model_maker.py --attempts=200 \
  CONFIG_FILE
```

will result in a more compact particle, as the one shown in Fig. 4.5.

The structure shown in Fig. 4.5 may look more convincing than the one of Fig. 4.4 for the case of a condensing particle, but it still shows important cavities. In principle, we could try shrinking the maximum radius to obtain a more compact structure. Generally speaking, small allowed radii will result in compact, roundish particles, while large allowed radii will produce fluffy and irregular structures. However, there is a limit on the amount of spheres with fixed radii that can enter within a given spherical volume without compenetration. If the radius is too small, the script may fail to randomly find a place to drop a new sphere and eventually give up trying. In this case, a warning message about the particle not using all the assigned spheres is printed. The default number of maximum random attempts per sphere used by `model_maker.py` is 100, but this can be changed with the `--attempts=N` command line argument shown in the example above. Such option, however, needs to be used with care, because increasing the number of random trials in an overly packed particle structure will slow down the execution, without solving much.

A more efficient strategy to improve the filling factor of a model particle is to define the type distribution of spherical monomers, taking into account the fact that the `model_maker.py` script drops them in the same order as they were defined. Thus, if, for instance, the script is instructed to place all the largest spheres first and, then, to proceed with the smallest ones, there are better chances that large inner cavities will be patched with one or more spheres. To obtain this effect, e.g. first using 25 large spheres and then 50 small ones, we can replace lines 38-39 with:

```
# Vector of sphere type identifiers (what type is each sphere)
sph_types   : [
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
```

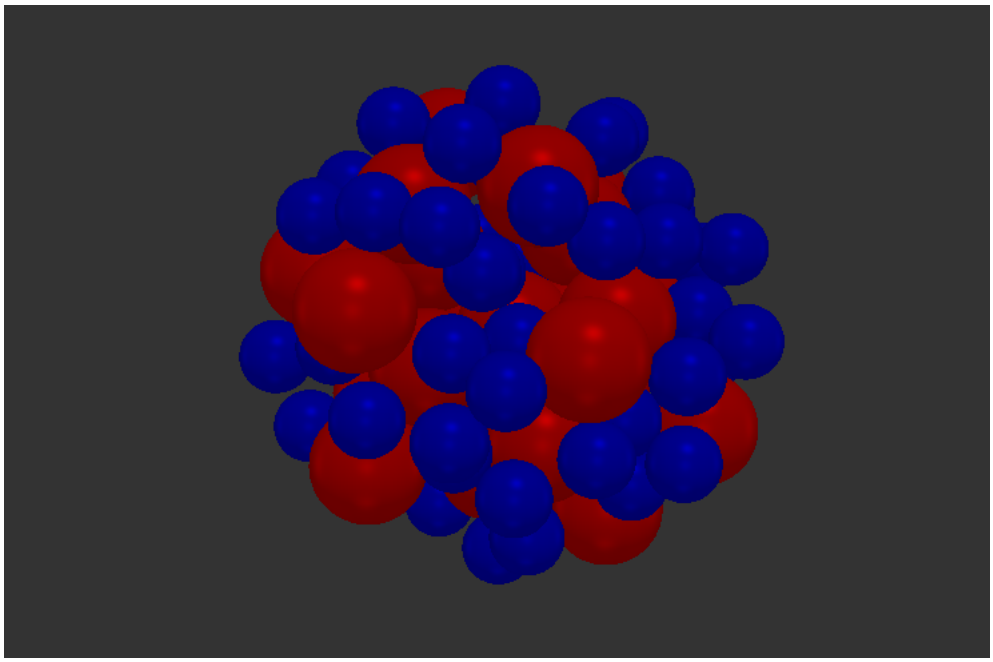

Figure 4.6: Same as Fig. 4.4 and Fig. 4.5, but with pre-assigned sphere types. In this realization, the `model_maker.py` script was instructed to drop all the large monomers first, subsequently adding the small ones. This strategy improves the filling factor of the particle.

```
  2, 2, 2
]
```

and line 51 with:

```
max_rad     : 6.0e-6
```

The result of these settings should look like the example shown in Fig. 4.6.

`model_maker.py` has no restrictions in the sorting order of the vector of sphere types (as instead it happens in the structure of the `DEDFB` files discussed in Section 2.5.1). Large spheres can be mixed with small ones at will. It is up to the user choosing the strategy that more closely models the desired situation. After generating the structure, the script will take care to sort all the information concerning the sphere types, finally writing them to the application configuration files in order of increasing type. For this reason, the type labeling among the `YAML` configuration file and the corresponding

DEDFB input file can show substantial differences. For example, recalling the restrictions on the sphere type vector in a DEDFB file discussed in Section 2.5.1, the pre-assigned type vector above would be written in the DEDFB file as a sequence of 25 characters '1', followed by 50 times the number '26'.

4.5 Computing a model

Using the edited model with pre-assigned monomer types, discussed in the previous section, the execution of the `model_maker.py` script reports the following log:

```
1  INFO: estimated matrix size is 4.83 Gb.
2  INFO: system supports up to 9 simultaneous processes on GPU.
3  INFO: only 3 GPU processes allowed, if using refinement.
4  INFO: model requires 6.5614GiB of host RAM.
5  INFO: system supports up to 30 simultaneous processes
6        (N.B.: not including overheads!)
7  INFO: the number of detected CPUs is 20.
8  INFO: smallest monomer radius is Rmin = 7.94328e-07m
9  INFO: largest monomer radius is Rmax = 1.25893e-06m
10 INFO: equivalent volume radius is Reqv = 4.21607e-06m
11 INFO: minimum encircling radius is Rcirc = 5.43680e-06m
```

Recalling that the configuration settings described in Section 4.4.3 declare the limiting resources that we want to set for each *MPI* process, `model_maker.py` extracts some information on the predicted resource requirements and compares it with the declared hardware limits. The first piece of information given in line 1 is the estimated size of the T-matrix. This is compared with the declared VRAM, to assess how many simultaneous calculations could be supported when using a GPU (line 2). Line 3 corrects this preliminary estimate to the one that should be considered, if using iterative matrix inversion refinement,[1] a feature that improves numerical accuracy of the solution, but requires 3 times more memory in the stage of matrix inversion. Lines

[1] We are not descending here in the details of numeric stability, but this is a problem that commonly arises when trying to solve systems which require to account for values separated by many orders of magnitude with a limited precision on their numeric representation. Interested readers may check the code's technical reference paper.

4 and 5 attempt to perform a similar resource requirement estimate for all the data stored by the host system to solve a single wavelength iteration. This estimate, however, does not take into account the overheads (line 6), which, depending on the optimization may be as large as a factor of 10. The following lines (from 8 to 11) are almost self-explanatory and they report, respectively, the radius of the smallest monomer in use, the radius of the largest one, the radius of a single sphere that would fill the same volume as the one of the model particle, and the radius of the smallest sphere that contains the whole particle.

The information on particle size and memory requirements are strictly connected. Indeed, a fundamental parameter that controls the size of the data structures is the maximum order of the calculation before the truncation of the field expansion, when working with the single monomers (internal order) and with the whole particle (external order). These values should be set based on the size parameters of the structures that need to be solved (i.e., the largest monomer within the particle, for the internal order, and the encircling sphere for the external one). A proper choice of these parameters is critical for the calculation setup, because setting an unnecessarily high order leads to increased resource consumption and risks of numerical instability, while falling short of the optimal value results in a non converging, unreliable solution. For single spheres and very simple models, the recommended choice of expansion orders can be obtained via the Wiscombe criterion, implemented by the `pywiscombe.py` script. However, for very complicated models, this criterion may not provide accurate answers and a more advanced procedure needs to be followed. The process of configuring the calculation for a target host system, like a computing farm, and executing it via the `SLURM` system is detailed in the next sections.

4.5.1 Configuring the execution

The most reliable way to configure a model calculation is to perform a *convergence test*. This is a test that replaces the particle with two minimal bracketing models, made by a single sphere, respectively corresponding to the largest monomer in use and to the smallest encircling sphere. If we run

the `pywiscombe.py` script on these two cases, using the shortest wavelength that our calculation needs to solve (i.e., the worst possible case, because the particle will have the largest size in units of wavelength), we will obtain a hint on the highest orders that could be naively expected to be necessary. For the case of the previous section, we would use:

```
WORK_DIR$ INSTALL_DIR/src/scripts/pywiscombe.py --wave=1.0e-6 \
  --rad=1.26e-6 --li
Suggested truncation order is Lmax = 18
WORK_DIR$ INSTALL_DIR/src/scripts/pywiscombe.py --wave=1.0e-6 \
  --rad=5.44e-6 --li
Suggested truncation order is Lmax = 50
```

The results of the Wiscombe test suggest that, working at a wavelength of $1.0\,\mu$m with our model particle, we could search for convergence on the largest monomer exploring field truncation orders up to 18 and for convergence on the whole particle exploring field expansion orders up to 50.

The convergence test can be performed by solving the case of a single sphere (i. e., using `np_sphere`) for the two bracketing sizes, starting from a low expansion order and then increasing up to 18, when testing the inner order with a sphere having $r_{sph} = 1.26\,\mu$m, and up to 50, when testing the outer order with a sphere of $r_{sph} = 5.44\,\mu$m. `np_sphere` is a light-weight calculation that can be solved by commercial hardware even at quite high orders. `NP_TMcode v0.10.5` features a *bash* shell script which repeats the single-wavelength calculation of `np_sphere` for all the truncation orders of interest. The script is a LINUX executable file named `convergence.sh` and located under the `src/scripts` folder. Its preamble reads as:

```
1  #!/bin/bash
2
3  # Script settings
4  l_start=2 # first order to test
5  l_end=20 # last order to test
6  wavelength="1.000E-06" # wavelength to test (use your shortest one)
7  dielec_real="2.6015926E+00" # real part of dielectric function
8  dielec_imag="3.8710688E-04" # imaginary part of dielectric function
9  radius=$1 # radius is read from command line
10 ompthreads="" # OMP threads to use (default is "1,NPROC")
11 outputfile="convergence.csv"
```

In this configuration, the script is ready to perform a convergence test on the inner order for a forsterite particle at the wavelength of 1 μm. If we wanted to test a different wavelength or another material, we would need to declare the desired wavelength in line 6 and the real and imaginary part of the corresponding dielectric function in lines 7 and 8 (the curious reader may look in the `DEDFB` file built on the basis of our example model and find out that the numbers in lines 7 and 8 are the real and imaginary part of the first dielectric function of the material, connected with the first wavelength of 1.0 μm). Since the script is ready for use with forsterite, we can simply run it with:

```
WORK_DIR$ INSTALL_DIR/src/scripts/convergence.sh 1.26e-6
```

The script takes as single command line argument the radius of the particle element that needs to be tested and then it autonomously runs `np_sphere`, writing the application output in a temporary folder, which is deleted after parsing the results for housekeeping purposes. The results of the script are saved in a plain ASCII file named according to the value of the `outputfile` parameter of line 11. Inspection of this file will show that the numerical results of the code (in this case the direction averaged cross-sections) will achieve good converge at `LM` = 10, becoming numerically stable at `LM` = 12 (before the Wiscombe based estimate of 18).

Performing the test on the external expansion order requires some editing of the script. Indeed, while the Wiscombe criterion may be as high as 50, the script, by default, only tests up to `LM` = 20 (line 5). We can change this limit by editing the `l_end` parameter:

```
l_end=50 # last order to test
```

and run again the script, using, this time:

```
WORK_DIR$ INSTALL_DIR/src/scripts/convergence.sh 5.44e-6
```

to test for the encircling sphere. Inspection of the new results will show that the calculation is already very stable with an expansion order of `LM` = 42, though minor effects can be appreciated up to `LM` = 45. The convergence trends for the internal and the external expansion order tests is further illustrated by Fig. 4.7. Based on the diagnostics of the figure, the settings of

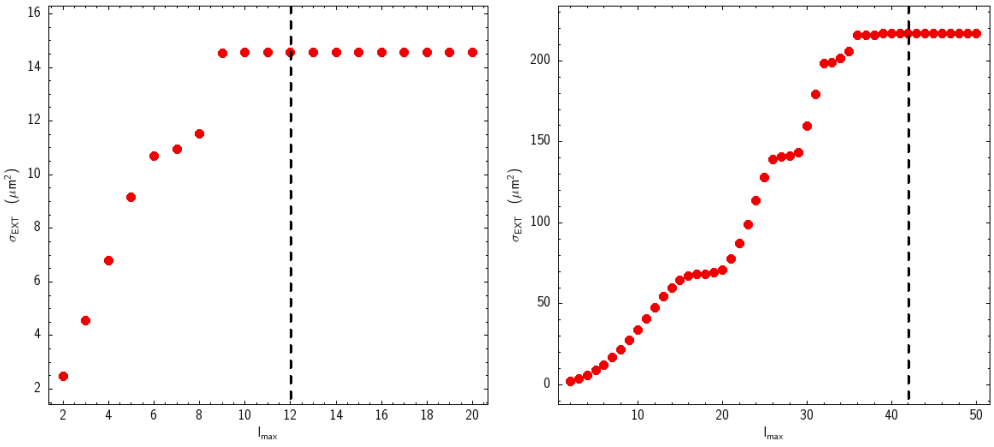

Figure 4.7: Convergence of the extinction cross-section for the internal field expansion order (inner test, left panel) and for the external field expansion order (outer test, right panel). The black dashed vertical lines indicate the numeric convergence order for the two cases (respectively, `LI` = 12, `LE` = 42). It can be appreciated how acceptable degrees of convergence can be obtained even with somewhat lower orders.

`li` and `le` in lines 94 and 96 of the model configuration example shown in Section. 4.4.3 are properly justified.

4.5.2 Running the calculation

The fundamental approach to run a model calculation has been illustrated in Section 3.4 for the case of a simple model on a local workstation. The example model that we have been discussing since Section 4.4.3 much more demanding hardware requirements and it is more suitable for being solved by one or more computing nodes via a batch job submission system, like `SLURM`. Assuming that `NP_TMcode` has been installed in a computing farm that uses `SLURM` and provides the dependencies to support calculations performed on the GPU (specifically, the `CUDA` and `cuBLAS` libraries, for NVIDIA based GPU architectures, and the `MAGMAv2` library), the proper way to run a calculation is to prepare a batch job script, using the resource requirement estimate information of the model configuration step, compared with

the hardware resources of the target computing nodes, to set up the best calculation strategy.

To make an example, we will consider a typical configuration for a calculation submitted to the computing farm hosted at the INAF - Astronomical Observatory of Cagliari. This facility features a `gpu` partition where each node is equipped with two AMD Epyc 7313 CPUs, supporting 64 computing cores in total, two NVIDIA A40 Ampère GPUs, 503 GiB of host RAM and 48 GiB of VRAM per GPU, summing up to a total of 96 GiB of VRAM. Recalling that `NP_TMcode` estimates its resource requirement on a *per-process* basis and that each *MPI* process gets a single assigned GPU, we design our parallel strategy to use 4 *MPI* processes, distributed among 2 nodes (thus employing 4 GPUs in total), and calibrating the execution in such a way that the simultaneous threads of each *MPI* process will not overflow the VRAM of the GPUs or the total RAM of the host system. The aforementioned strategy can be implemented in a `SLURM` batch script, featuring the following configuration:

```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=2
#SBATCH --cpus-per-task=32
#SBATCH --partition=gpu
#SBATCH --job-name="forst_n75"
#SBATCH --output=model.out
#SBATCH --error=model.err
#SBATCH --gpus-per-node=2
#SBATCH --time=12:00:00
```

The above code is standard `SLURM` batch syntax to request full access to the `gpu` partition nodes featuring the hardware that we discussed previously. After the resource requests for the scheduler, the script may proceed as follows:

```
# Print information about the job
echo "JOB ID is $SLURM_JOB_ID"
echo "Host node is $SLURMD_NODENAME"

# Set up the environment
module load magma/2.7.1/cuda-12.1/mkl-ilp64-2023.0.0/gnu-13.1.0
module load hdf5/1.14.0/openmpi-4.1.5-gnu-13.1.0
```

Here, lines from 11 to 13 are used to print information on the job ID and the node name, as identified by the SLURM system, while lines 16-17 load the modules that feature the libraries which were linked to the code and their dependencies

```
18 export JOBSCRIPTDIR=[...] # location of the batch script
19 export JOBOUTPUTDIR=$JOBSCRIPTDIR/results
20 export BUILDDIR=INSTALL_DIR/build
21 export DATADIR=$JOBSCRIPTDIR
22 export JOBLOCALDIR=[...] # working location on scratch or node local \
23   file system
24 export CUDA_MPS_LOG_DIRECTORY=$JOBLOCALDIR/cuda_mps_logs
25 export CUDA_MPS_PIPE_DIRECTORY=/tmp/nvidia-mps
26 export OMP_NUM_THREADS=2,16
27 export NP_RESULT_DIR="."
```

This section defines the working folder structure and some environment variables to control an optimized parallel execution. Here INSTALL_DIR must point to the location where the code was installed on the farm, while the JOBSCRIPTDIR and the JOBLOCALDIR must be set by the user to point, respectively, to the folder where the batch script is stored (which will finally get a copy of the results) and to a scratch work space where the computing nodes can perform their I/O operations. Lines 24 and 25 are used by the CUDA multi-process daemon, a service that creates a single context for all the processes that communicate with a GPU. Instead, particular attention must be paid to the choice of the thread parallelization strategy, controlled by the OMP_NUM_THREADS variable of line 26. In our case, the configuration script estimated that up to 3 simultaneous threads could be handled by each GPU, while a quick estimate of the overheads points to a host memory consumption of the order of 190 GiB per *MPI* process. The described system would safely support 3 threads in the first level of *OpenMP*, but this would leave only up 10 threads for the second hierarchy level, failing to fully employ the available computing cores. Therefore, for a safer resource requirement and a better work balance among host system and GPUs, our choice here is to set 2 simultaneous wavelength threads as the first level of the *OpenMP* parallel hierarchy and 16 work sharing threads for the second. Line 27 simply sets where the code will write its output files (i. e. in the local job working

directory, in our example).

With the parallel environment set up, the script needs to create the necessary folder structure and move into the working directory. This is achieved with:

```
28 mkdir -p $JOBOUTPUTDIR
29 mkdir -p $JOBLOCALDIR
30 mkdir -p $CUDA_MPS_LOG_DIRECTORY
31 # Now go into the local dir to run C++ codes there
32 cd $JOBLOCALDIR
```

If the system supports the `nvidia-cuda-mps-control` to create a common context for the execution of multi-process calculations, this would be activated with:

```
33 # Start MPS daemon
34 echo "Starting MPS"
35 nvidia-cuda-mps-control -d
```

The following step would be to copy the configuration input files for the calculation into the working directory and run the code. Assuming that the input files were constructed by the `model_maker.py` script, with the naming conventions defined by lines 23 and 25 of the configuration example presented in Section 4.4.3, and that they were placed in the same directory where the batch job script is stored, this would be obtained with:

```
36 # Copy the data files in the job local directory
37 cp $DATADIR/DEDFB_rnd_loose DEDFB
38 cp $DATADIR/DCLU_rnd_loose DCLU
39
40 # Actually run np-tmcode
41 echo "Executing np_cluster..."
42 mpirun -n 4 $BUILDDIR/cluster/np_cluster DEDFB DCLU $NP_RESULT_DIR
```

In case the calculation ran under the `nvidia-cuda-mps-control` common context, the service needs to be stopped with:

```
43 # Stop MPS daemon
44 echo "Stopping MPS"
45 echo quit | nvidia-cuda-mps-control
```

Finally, the results of the calculation should be copied back to the chosen job folder and the local working space should be cleaned, closing the execution

with a job completion message:

```
46 # Copy the results to JOBOUTPUTDIR
47 echo "Copying back the results and cleaning local directory."
48 cd $JOBOUTPUTDIR
49 cp -r $JOBLOCALDIR/* .
50 rm -rf $JOBLOCALDIR
51
52 echo "Job finished."
```

4.5.3 Checking the results

With the batch job configuration illustrated in the previous section, the folder where the job script is located would be populated with the following contents:

- `DEDFB_rnd_loose`: the first configuration file;
- `DCLU_rnd_loose`: the second configuration file;
- `model.err`: a log of all the messages sent to `stderr` during the job execution;
- `model.out`: a log of all the messages sent to `stdout` during the job execution;
- `results`: a folder containing the calculation results,

together with the batch script itself, of course. The first operations to perform, in order to verify the sanity of the run would be a check of the contents of the `model.err` and `model.out` files. The first one is expected to contain some logging messages from the module loading system and nothing else. If any additional messages appear in it, this is probably not good news. Conversely, the `model.out` file will contain all the logging messages generated by the code and by the batch job script commands. In case the code ran smoothly, the last lines of this file will read like:

```
INFO: Calculation lasted 2408.798311s.
Finished: output written to ./c_OCLU
Stopping MPS
```

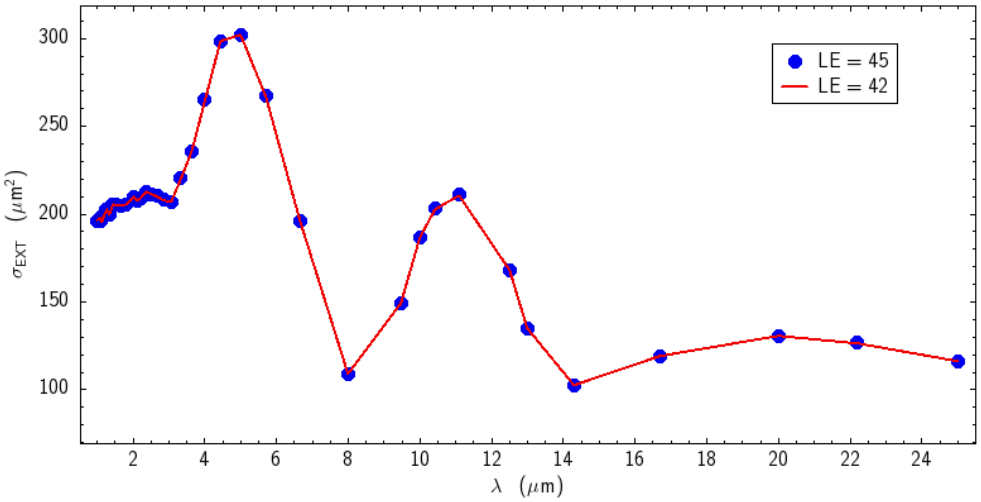

Figure 4.8: Extinction cross-section σ_{EXT} as a function of wavelength for the model particle described in Sections 4.4.3 and 4.5.2. The red continuous line represents the recommended calculation carried out using an external order setting of `le=42`. The blue points show a corresponding calculation carried out at `le=45`, to show that the results have converged and no further increase of the orders in use is needed for this model on the plotted wavelength range.

```
Copying back the results and cleaning local directory.
Job finished.
```

Otherwise, the last messages can be used to understand where the code was able to get, before stopping.

Assuming that the aforementioned preliminary tests gave the desired response, the following step would be to enter the results folder and to parse the output files to produce to extract plots of relevant quantities as function of wavelength. With our configuration, this can be done with:

```
SCRIPT_DIR$ cd results
results$ INSTALL_DIR/src/scripts/parse_output.py --in c_OCLU \
  --out results_li10_le42 --app=CLU
```

This will produce a set of comma separated value data files, that we chose to name `results_li10_le42*`, to keep track of the calculation orders that

we used. Plotting the extinction cross-section for this model particle should produce a result like the one shown in Fig. 4.8. To further test the reliability of the results, in terms of convergence, we repeated the same calculation, but with a slightly larger external expansion order of `le=45`. This can be tested rather easily, because increasing the external order of the calculation does not increase the pressure on the GPU, while only producing a secondary effect on the host RAM (where we chose to play a safe parallelization strategy). As clearly demonstrated by the figure, the two solution are totally consistent.

4.6 Known errors

There are a few common conditions that can prevent `NP_TMcode` from successfully perform a calculation. In this cases the code may either stop with a log message, containing information on what went wrong, or throw an exception (both conditions result in any parallel run to be aborted). If, however, the problem arises due to the calculation hitting the hardware limits of the host system, the result is implementation-dependent. An over-allocation of the GPU VRAM will probably lead to some exception condition. A similar condition on the host system mey result on the process being killed by the out-of-memory watch system or the host system to crash. For this reason, it is always a good idea to check the policy of the host system and to adopt a safe parallel strategy. Another condition that produces abnormal termination is the attempt to write to a non-existing (or non-writeable) output path.

To help users and developers having a better idea of what might have caused an abnormal termination, `NP_TMcode` defines the following exception classes (all derived from standard *C++* `std::exception` base class):

- `ListOutOfBoundsException`: this is thrown if attempting to access an element out of the boundaries of a dynamic list (dynamic lists are used in place of *C++* `std::vector<>` objects for some configuration operations where full control over memory access is preferred with respect to runtime performance). It can happen as the consequence of some ill-formed configuration files.

- `ObjectAllocationException`: this is thrown if the allocation of some data structure fails and the code realizes hit before the operating system does. It is generally a signal that the decalred hardware resource limits were violated.

- `OpenConfigurationFileException`: self-explanatory. The code attempted to open a non accessible configuration file (wrong file name, non readable file system, ...).

- `MatrixOutOfBoundsException`: this is thrown if an invalid access to a matrix element is attempted. This type of error has the purpose of categorizing developing errors on optimized matrix operations. It should never occur to a normal user.

- `UnrecognizedConfigurationException`: this exception is thrown whenever the code detects an invalid combination of settings in the configuration files. It prevents the code from running numerically valid, but physically meaningless calculations (e.g., overlapping spheres, non logically sorted layers, ...).

- `UnrecognizedFormatException`: this exception is thrown when asking code output in an unsupported format (something different from `HDF5`, `LEGACY`, and, in some cases, `ASCII`). It can only happen to developers.

- `UnrecognizedOutputInfo`: this is thrown if attempting to generate output information for a different application than one of those included in the software package. Such an event should never occur to a normal user, but can happen to code developers. It is defined to categorize errors connected with the development of output modules.

- `UnrecognizedParameterException`: this exception is thrown when an internal code function attempts access to an unregistered parameter calling it by name. It can only happen to developers.

In case any of the above exceptions occurs, the code will stop, signaling an abnormal termination, providing an explanatory message and a stack trace. The implementation of thorough exception handling strategies is part

of the `NP_TMcode` development plans and it will addressed in more advanced package releases.

Acknowledgments

Supported by Italian Research Center on High Performance Computing Big Data and Quantum Computing (ICSC), project funded by European Union - NextGenerationEU - and National Recovery and Resilience Plan (NRRP) - Mission 4 Component 2 within the activities of Spoke 3 (Astrophysics and Cosmos Observations). The authors would like to acknowledge OpenACC-Standard.org for their support. The `NP_TMcode` software is the parallel implementation of the T-matrix formalism, currently under development at INAF - Astronomical Observatory of Cagliari, on the basis of the codes originally written by F. Borghese, P. Denti and R. Saija.

Bibliography

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D., 1999. LAPACK Users' Guide. Third ed., Society for Industrial and Applied Mathematics, Philadelphia, PA.

Borghese, F., Denti, P., Saija, R., 2007. Scattering from Model Nonspherical Particles. Springer, Heidelberg. doi:10.1007/978-3-540-37414-5.

Bosma, W., Cannon, J., Playoust, C., 1997. The Magma algebra system. I. The user language. J. Symbolic Comput. 24, 235–265. URL: http://dx.doi.org/10.1006/jsco.1996.0125, doi:10.1006/jsco.1996.0125. computational algebra and number theory (London, 1993).

Dagum, L., Menon, R., 1998. Openmp: an industry standard api for shared-memory programming. Computational Science & Engineering, IEEE 5, 46–55.

Draine, B.T., Flatau, P.J., 1994. Discrete-Dipole Approximation For Scattering Calculations. Journal of the Optical Society of America A 11, 1491–1499. doi:10.1364/JOSAA.11.001491.

Mie, G., 1908. Beiträge zur Optik trüber Medien, speziell kolloidaler Metallösungen (contributions to the optics of polluted media, especially colloidal metal solutions). *Annalen der Physik* 330, 377–445. doi:10.1002/andp.19083300302.

Mishchenko, M.I., Travis, L.D., Mackowski, D.W., 1996. T-matrix computations of light scattering by nonspherical particles: a review. *Jour-*

nal of Quantitative Spectroscopy and Radiative Transfer 55, 535–575. doi:10.1016/0022-4073(96)00002-7.

Palik, E.D., 1991. "Handbook of optical constants of solids II". ACADEMIC PRESS, INC., Cambridge. doi:10.1016/B978-0-08-055630-7.50001-8.

Purcell, E.M., Pennypacker, C.R., 1973. Scattering and Absorption of Light by Nonspherical Dielectric Grains. The Astrophysical Journal 186, 705–714. doi:10.1086/152538.

Taflove, A., Hagness, S.C., 2005. Computational electrodynamics: the finite-difference time-domain method. 3rd ed., Artech House, Norwood.

Tazaki, R., Tanaka, H., 2018. Light Scattering by Fractal Dust Aggregates. II. Opacity and Asymmetry Parameter. *The Astrophysical Journal* 860, 79. doi:10.3847/1538-4357/aac32d, arXiv:1803.03775.

The MPI Forum, C., 1993. Mpi: a message passing interface, in: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing, Association for Computing Machinery, New York, NY, USA. p. 878–883. URL: https://doi.org/10.1145/169627.169855, doi:10.1145/169627.169855.

Waterman, P.C., 1971. Symmetry, Unitarity, and Geometry in Electromagnetic Scattering. *Physical Review D* 3, 825–839. doi:10.1103/PhysRevD.3.825.

Wiscombe, W.J., 1980. Improved Mie scattering algorithms. Applied Optics 19, 1505–1509. doi:10.1364/AO.19.001505.

Yee, K.S., 1996. Numerical Solution of Initial Boundary Value Problems Involving Maxwell's Equation in Isotropic Media. IEEE Transactions on Antenna Propagation 14, 302–307. doi:10.1109/TAP.1966.1138693.