



Finanziato
dall'Unione europea
NextGenerationEU



Ministero
dell'Università
e della Ricerca



Italiadomani
PIANO NAZIONALE
DI RIPRESA E RESILIENZA



Nano-particle Transition Matrix code

Release Notes

Report for M10a.02

G. La Mura, G. Mulas

March 2025

Contents

Scope of the document	2
1 Aim of the project	3
1.1 Introduction	3
1.2 Project break-down	4
1.3 Project status	4
1.4 Distribution	5
2 Release description	6
2.1 Parallel implementation	6
2.2 New release features	7
2.3 Code performance	8
3 Instructions for testing	12
3.1 Introduction	12
3.2 Testing with local builds	12
3.2.1 Building the code	12
3.2.2 Execution of the <code>sphere</code> case	13
3.2.3 Execution of the <code>cluster</code> case	14
3.2.4 Execution of the <code>inclusion</code> case	16
3.3 Testing with <code>docker/singularity</code>	17
3.3.1 Docker	17
3.3.2 Singularity	18
3.4 Comparing results	19
3.5 Model construction with <code>model_maker.py</code>	20

Scope of the document

This document is provided along with the M10a release of the NP_TMcode suite as a quick report on the status of the project. The aim of this document is to give an overview of the project goals, to provide a quick guide to navigate the progress achieved with respect to milestones and to give a set of fundamental instructions to perform tests of the code functionality. More detailed documentation, explaining the structure of the code and the role of its main components (data structures, functions and variables) is given in the form of `doxygen`-handled inline documentation, README files and wiki pages.

This document is organized in chapters:

- Chapter 1 presents the general scope of the project and a description of its milestones;
- Chapter 2 provides a description of the current code release and discusses how the project guidelines were implemented;
- Chapter 3 finally gives instructions on how to test the code release in its current form.

The contents of this document update the information included in the *Release Notes* of M7.00, M8.03 and M9.01, also included in the distribution. The introductory discussion presented in Chapter 1 is similar to the one given in the previous releases. The contents of Chapters 2 and 3 assume that the reader is aware of the discussion presented in the release notes of M7.00, M8.03 and M9.01.

NOTE FOR REVIEWERS: A new feature introduced with NP_TMcode-M10a is a wiki document section, providing an overview of the status of the project with respect to the scheduled milestones, in the form of a progress bar scheme and a target checklist. This resource is publicly available at:

https://www.ict.inaf.it/gitlab/giacomo.mulas/np_tmcode/-/wikis/home

Chapter 1

Aim of the project

1.1 Introduction

This project aims at implementing High Performance Computing (HPC) strategies to accelerate the execution of the Nano-Particle Transition Matrix code, developed by Borghese et al. (2007), to solve the scattering and absorption of radiation by particles with arbitrary geometry and optical properties. The goal is to migrate the original code, written in *FORTRAN 66*, to a modern programming language, able to access new hardware technologies, thus substantially reducing the amount of time required for calculations through the use of parallel code execution handled by multi-core computing units.

The problem of radiation absorption and scattering has a large variety of applications, ranging from Astrophysics of the interstellar medium (ISM) and (exo)planetary atmospheres, all the way to material investigation through optical techniques and nano-particle handling by means of optical tweezers. In spite of the large impact of such problems on both scientific and technological applications, the theoretical framework of radiation/matter interactions has mostly been treated under the assumption of simplifying conditions, such as plane-wave radiation fields interacting with spherical particles.

Dealing with more realistic cases is only possible through numerical calculations. These can be broadly distinguished in the *Discrete Dipole Approximation* (DDA) based approach (Draine & Flatau 1994), which subdivides a general material particle in a properly chosen combination of dipoles, thereby solving their interaction with the radiation field, and the *Transition Matrix* (T-matrix) solutions (Borghese et al. 2007) which, on the contrary, take advantage from the expansion of the radiation fields in multi-polar spherical harmonics to create a set of boundary conditions that connect the properties of the incident and of the scattered radiation at the surface of the interacting particle layers. In the latter case, the particle is approximated as a collection of spherical sub-particles that acts as a mathematical operator connecting the properties of the incident and scattered field.

The T-matrix method has the main advantage of creating a unique link between the incident and the scattered radiation fields, offering a solution that is valid for any combination of incident and scattered directions and at various distance scales, from within the particle itself, all the way up to remote regions. While the T-matrix itself is computed numerically, it then provides an analytical expression of incident and scattered fields for any arbitrary incident wave, enabling (relatively) easy arbitrary averages over combinations of relative orientations of incident fields and complex scattering particles. In particular, it is easy to account for different partial alignment conditions, with no need to repeat the calculation of the T-matrix.

Conversely, DDA calculations need to be entirely solved for each combination of incident and scattered radiation fields and they require a different description of the particle, depending on the dimension scale they apply to. As a consequence, the T-matrix method is largely preferable for the investigation of all types of integrated effects, thus naturally covering also the dynamic and thermal effects of the radiation-particle interaction.

While T-matrix based solutions are ideal to solve the scattering problem in all circumstances where multiple scales and directions need to be accounted for, the calculation of the Transition Matrix for realistic particles is a computationally demanding task. The `NP_TMcode` project takes on the challenge of porting the original algorithms to modern hardware, in order to substantially reduce the computational times and allow users to model more complicated and realistic particle structures, and/or large populations of different particles, in shorter execution times.

1.2 Project break-down

We can broadly divide the project in three main stages, namely corresponding to:

1. code porting to C++
2. implementation of parallel algorithms
3. **deployment of general radiation/particle interaction solver**

These three stages are associated with a set of Key Performance Indicators (KPI), consisting in code releases with enhanced computational abilities. The current project stage provides a configurable and scalable parallel implementation of the calculation, built on the basis of the *C++* ported algorithms, that is able to adapt to the hardware and software capabilities of the host system where it is executed and to store the results in the standard HDF5 binary format.

1.3 Project status

We refer to the current release of the project as `NP_TMcode-M10a.02`, since it includes updates on the documentation and the distribution of the code that address the targets of

the project Milestone 10a (namely, a radiation/particle interaction solver, with improved model definition, able to deal with particle models encompassing inclusions, using different types of multi-thread and multi-core acceleration technologies and writing the results in **HDF5** format). The code is distributed through a public **gitLab** repository, under GNU General Public License Version 3. The current implementation provides the following new features with respect to **NP_TMcode-M9.01**:

- the implementation of a parallel solver for the case of particle with inclusions;
- the possibility to build model particles formed by clusters of multi-layered spheres;
- the implementation of **HDF5** format output for all the computed values;
- the inclusion of inline documentation for all the quantities written to output;
- the implementation of a model creation helper script;
- an expanded set of test cases, diagnostic tools and debug options.

In addition to these specific features, the use system resources has been further optimized by reducing the number of hard-drive I/O calls, defining more effective memory structures to manage the communication of data among threads and processes and adding a new sanity check to the continuous integration development pipeline, to ensure proper handling of host system resources.

This set of newly implemented features are intended to enhance the user's ability to build the code either on personal workstations or on shared computing systems with different hardware characteristics. In addition, the code optimization features allow for the calculation of higher complexity models with relatively smaller computational effort, leading to the execution of more advanced scientific runs.

1.4 Distribution

The **NP_TMcode** project is developed and distributed at the following **gitLab** public repository:

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

Discussion about the code performance and features is also possible through the **gitHub** project release portal:

https://github.com/GLAMURA81/NP_TMcode_release

Chapter 2

Release description

2.1 Parallel implementation

NP_TMcode-M10a.02 is the fourth official release of the *Nano-particle Transition Matrix Project* code,¹ funded under the project *CN-HPC, Big Data and Quantum Computing CN_00000013, Spoke 3 "Astrophysics and Cosmos Observations"* (CUP C53C22000350006). The aim of this release is to implement the solution of the case of a particle with inclusions and to improve the input model definition and the output storage in standard binary format.

The features introduced in this release followed two main guidelines:

1. All newly introduced features must preserve the consistency with pre-computed test cases.
2. They must result in a smaller work-load on the user side.

In order to grant for code setup flexibility, this release includes a configuration script and a model generator script. When executed, the configuration script runs a set of standard tests to detect the host system hardware and software capabilities to choose the most convenient compiler options. As a consequence, the user is no longer required to manually set compiler flags, although the possibility to customize the compilation by manually setting various compiler variables is still available in the form of configuration options. More details on how to configure and use the code to access these possibilities are given in Chapter 3 of this document.

¹Check https://www.ict.inaf.it/gitlab/giacomo.mulas/np_tmcode/-/releases for tracking all the code releases.

2.2 New release features

Compared with the previous release, NP_TMcode-M10a.02 has some fundamental improvements. These mostly concern the possibility to handle new particle cases in parallel execution, to write all the results in a standard binary format and to build particle models in an easier and more descriptive manner. In addition to these new code functionalities, NP_TMcode-M10a.02 also includes improved inline documentation and a new section of wiki documents to track the progress of the project.² Many of the code optimizations are automatically managed, meaning that the code will scan the system to detect the available resources and derive the best set of compilation flags. If necessary, some of the optimization capabilities can be manually overridden, by means of a set of configuration options that control the code optimization level, the amount of debug information and the use of external libraries, which may reside in different locations, with respect to the system defaults. All such features are detailed in the configuration script run-time help system, which can be inspected by issuing:

```
~/build> ./configure --help
```

from the `np_tmcode/build` folder.

The main novelties that NP_TMcode-M10a.2 offers, with respect to the previous releases are:

1. The application to solve the case of a spherical particle with inclusions.
2. The possibility to use iterative matrix inversion refinement to improve the numerical accuracy of results.
3. The production of all the code output in HDF5 binary format (together with the legacy formatted text format).
4. A model generating script that relieves the user from the task of formatting the input according to the legacy requirements, introducing a more descriptive standard based on YAML.

The introduction of these features greatly reduces the complexity of the creation of new models, in addition to those included in the release bundle. The revision of the I/O management system also improved the communication of data among different threads and processes, allowing for the transition of parallel algorithms from an almost exclusive dependence on the MPI standard (characteristic of NP_TMcode-M9.03) to a wider use of the OpenMP framework in NP_TMcode-M10a.02. This particular feature introduces some degree of performance improvement and allows for the execution of parallelized algorithms by a larger set of multi-threaded systems, such as commercial laptops, without necessarily requiring the user to provide a full MPI-compliant compiler set.

²https://www.ict.inaf.it/gitlab/giacomo.mulas/np_tmcode/-/wikis/home

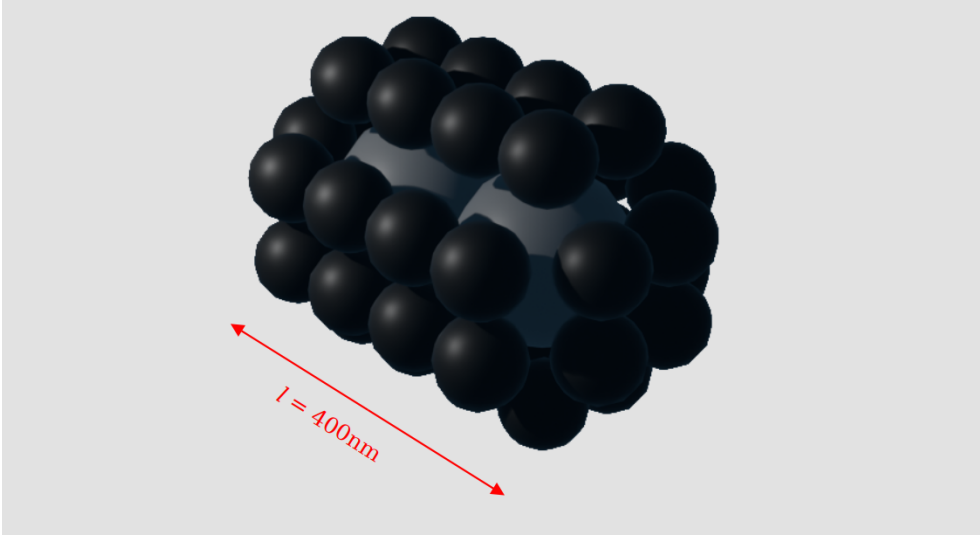


Figure 2.1: Illustration of the model particle used to evaluate the code performance. The particle is made up by a core of silicate materials (grey spheres) surrounded by a mantle of amorphous carbon (black spheres). The particle size along its largest extension is 400 nm.

2.3 Code performance

The code performance for Milestone 10a has been evaluated by applying the code to the same particle model used in Milestone 9 and illustrated in Fig. 2.1. The particle is composed by a combination of 42 spherical monomers, arranged in a core-mantle structure. Such model is intended as a starting point to test the code ability to deal with non-spherically symmetric clusters of units composed by different materials, therefore allowing for a more accurate representation of the conditions that can occur in physical particles such as interstellar dust grains. More in detail, the particle core is represented by 2 spherical monomers, each with radius $\rho_{core} = 68.63$ nm and composed by astronomical silicates, while the mantle is made of 40 spherical monomers, with radius $\rho_{mantle} = 34.31$ nm and assuming amorphous carbon as the composing material. The total particle size, along its largest extension is $l = 400$ nm and the scattering problem is solved for 181 different wavelengths, ranging between $100\text{ nm} \leq \lambda \leq 1\text{ }\mu\text{m}$, in steps of 5 nm. The optical properties of the materials were assumed to be the ones reported by Draine & Lee (1984) for astronomical silicates and by Palik (1991) for amorphous carbon.

Since this case is already too advanced to obtain reference results from the original implementation on reasonable times, the code can no longer be tested by comparing the results of the original implementation with those provided by the NP-TMcode project.

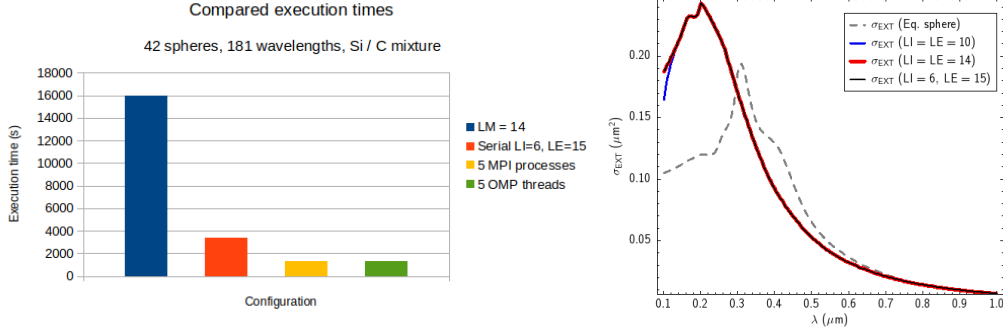


Figure 2.2: Calculation execution times (left panel) and particle extinction cross-section as a function of wavelength (right panel) for different problem configurations. The code performance is evaluated by means of the reduction in execution time and of the stability of the result with respect to the most computationally intensive task.

To solve the problem, instead, we adopted the approach used for Milestone 9, based on the execution of the same model on different hardware architectures, using different configuration parameters. We used as a reference case the calculation of the particle’s extinction cross-section as a function of wavelength for a fixed maximum field expansion order of $l_{MAX} = 14$ and we investigated the effects of adopting different field expansion orders, both on the execution times and on the stability of the final output. We compared our calculations with the results predicted for the equivalent spherical particle, i.e. a particle with the same mass and composition as the model presented in Fig. 2.1, but in the assumption of spherical geometry. This configuration, which is represented as a gray dashed line in the right panel of Fig. 2.2, is a useful test because the scattering problem in spherical geometry can be solved very quickly and, thanks to the *Rayleigh approximation*, we expect all physical solutions to converge to the same values at long wavelengths, while large deviations are predicted at shorter ones.

To evaluate the performance of the code and the numerical stability of the results, we ran the calculation on two hardware configurations. The first one, named configuration A, is a single node from the GPU partition of computing facility at the Astronomical Observatory of Cagliari, featuring 2 AMD EPYC 7313 processors, each with 16 hardware cores, which appear as 32 “logical” cores due to hyperthreading, for a total of 64 “logical” CPU cores, 512 Gb RAM and 2 NVIDIA A40 GPUs with 45 Gb of dedicated RAM. The second one, named configuration B, is an ASUS Zenbook laptop computer, with 32 Gb of RAM, 20 CPU cores and a NVIDIA GeForce RTX 4060 Laptop GPU with 8 Gb of dedicated RAM.

The complete solution of the scattering problem with fixed maximum field expan-

Table 2.1: Execution times of the test model particle for different hardware and field expansion order configurations. The table columns contain, respectively, the system configuration, the size of the T-matrix, the number of simultaneously working MPI processes, the number of **OpenMP** threads, the internal field maximum expansion order, the external field maximum expansion order and the calculation execution time in seconds.

CONFIG.	T-matrix size	N_{proc}	$N_{threads}$	l_{INT}	l_{EXT}	EXEC. TIME
A	$[18816 \times 18816]$	8	1	14	14	15963.02 s
B	$[10080 \times 10080]$	3	1	10	10	12851.70 s
B	$[4032 \times 4032]$	1	1	6	15	3370.58 s
B	$[4032 \times 4032]$	5	1	6	15	1348.13 s
B	$[4032 \times 4032]$	1	5	6	15	1293.96 s

sion order $l_{MAX} = 14$ involves the inversion of a $[18816 \times 18816]$ elements complex matrix, which, when represented in double precision, requires 5.3 Gb of memory space to be stored. We, therefore, handled this case using the hardware configuration A to obtain our reference cross-sections, because it is the only one able to simultaneously work on multiple wavelengths. However, thanks to the fact that the particle's spherical components are smaller than the whole particle and can therefore be solved with a lower field expansion order than the one used for the whole particle, we took advantage of the possibility to distinguish among the internal maximum field expansion order l_{INT} and the external maximum field expansion order l_{EXT} to investigate the effects of using different calculation orders on the execution time and the result stability. As it turns out, the size of the T-matrix is controlled by l_{INT} , while still preserving the accuracy of the solution using a higher value of l_{EXT} (Iatì et al. 2004). The results of our tests are illustrated in Fig. 2.2 and further detailed in Table 2.1.

The situation depicted in Fig. 2.2 confirms that the choice of the field expansion order has a critical role. While we observe that all the computed models converge to the *Rayleigh* solution in the long wavelength regime, as expected, at shorter wavelengths there are large deviations from the behavior predicted in spherical symmetry. Here we see that, although all calculations converge to very similar solutions, lowering the maximum field expansion order l_{MAX} from 14 to 10 leads to a performance improvement at the cost of loss of precision at the shortest wavelengths (i.e. the wavelengths where the use of high maximum field expansion orders are more critically required). Conversely, the use of distinct maximum field expansion orders for the internal and external calculations implies a strong reduction of execution time, while fully recovering the precision achieved in the

highest order calculation. In addition to these results, which had already been attained at the conclusion of Milestone 9, `NP_TMcode-M10a.02` was further tested by executing the most lightweight calculation using `OpenMP` parallel threads instead of simultaneous `MPI` processes. This new option led to slightly better performance, extending the possibility to execute the parallel implementation of the code also to host systems that do not provide `MPI`-compliant compilers and runtime capabilities.

Chapter 3

Instructions for testing

3.1 Introduction

Testing of NP-TMcode-M10a can be achieved through two main approaches. The first is to obtain a local install, building the source code on the user's own machine. The second is to use a pre-built image, choosing between the `docker` and the `singularity` implementations. § 3.2 deals with testing the code release on a local machine. Detailed instructions on how to use pre-built images, instead, are given in § 3.3.1.

3.2 Testing with local builds

The first operation needed to build and execute the code is to replicate the project release from the *gitLab* repository. We assume that this step has been already performed, since the present document is distributed as part of the release bundle. The following steps, therefore, are just building and execution.

3.2.1 Building the code

To build the code, the user needs a set of compilers and some libraries. The recommended setup includes an up to date installation of the *GNU Compiler Collection* (`gcc`), of the *GNU make* builder, a *FORTTRAN* compiler (again, the recommended option is to rely on *GNU's gfortran*) and the *doxygen* document manager. An optional dependency is a working *L^AT_EX* distribution with recommended package setup, in order to build the full PDF documentation. If the aforementioned system requirements are met, to build the code after having retrieved the sources from gitlab, the user simply opens a shell terminal, goes into the `np_tmcode/build` folder:

```
~/np_tmcode> cd build
```

and then follows the standard steps of configuring and compiling the code with:

```
~/build> ./configure
```

and:¹

```
~/build> make
```

The compiler flags that were used to toggle optional features up to version M8.03 have been replaced by a set of configuration options, which can be inspected by issuing:

```
~/build> ./configure --help
```

from the `build` folder.

The build process will take care of building all of the *FORTRAN* and *C++* codes, placing the relevant binary files in a directory structure based in the `build` folder, located at the same level of the `src` folder in the `np_tmcode` directory structure.

If desired, one can optionally build the inline documentation of the code by issuing the command:

```
~/build> make docs
```

This will generate a folder named `doc/build` under the `np_tmcode` directory, with two additional sub-folders, respectively named `html` and `latex`. The `html` folder contains a browser formatted version of the inline code documentation, starting from a file named `index.html`. The `latex` folder, on the other hand, contains the instructions to build a PDF version of the documents, using \LaTeX , by issuing the further `make` command (*after* the previous `make docs` step), from the same directory:

```
~/build> make -C ../doc/build/latex
```

3.2.2 Execution of the sphere case

Once the build process has been completed, the code is ready to be run. A number of example test cases are available, provided together with the code. The configuration files and the corresponding expected output files, as produced by the legacy reference *FORTRAN* code, are collected in a folder named `test_data` under `np_tmcode`. To run the *FORTRAN* code on the case of the single sphere, move to the `sphere` binary folder:

```
~/build> cd sphere
```

¹In case the testing system supports multi-core architecture, it is possible to issue `make -j` instead of simply `make`. This will build the various code sections in parallel, reducing the compilation time.

then run the *FORTRAN* configuration program:

```
~/sphere> ./edfb_sph
```

The `edfb_sph` program looks for the problem configuration data in a file named `DEDFB`² and located in the `test_data/sphere` folder, then it writes a formatted output file named `OEDFB` and a binary configuration file named `TEDF` in the current working directory. After checking for the existence of these files, the calculation of the scattering process, according to the *FORTRAN* implementation, can be executed by issuing:

```
~/sphere> ./sph
```

The `sph` program gets its input from a file named `DSPH` in the test data folder. Its execution provides essential feedback on the status of the calculation and writes the results in a binary file named `TPPOAN` and a text file named `OSPH`.

The calculation executed in *FORTRAN* can be replicated in *C++* simply by invoking:

```
~/sphere> OMP_NUM_THREADS=1 ./np_sphere
```

The `np_sphere` program, when executed without arguments, adopts the default behaviour of looking for the same input data as the *FORTRAN* code and writing the same type of output files, but prepending a `c_` prefix to the name of its output file. Optionally, it can be run as:

```
~/sphere> ./np_sphere PATH_TO_DEDFB PATH_TO_DSPH OUTPUT_FOLDER
```

to specify input files and output directory other than the default ones.³

After the execution of the *FORTRAN* and the *C++* versions, their final outcomes can be compared by checking the contents of the `OSPH` and the `c.OSPH` files (see § 3.4 for suggestions on how to compare files).

3.2.3 Execution of the cluster case

Execution of the cluster calculation is very similar to the sphere case. The first step is to move to the `cluster` folder:

```
~/sphere> cd ../cluster
```

²Explanations on the structure of the input data file are given in the `README.md` file stored in the `test_data` folder.

³`np_sphere` is now able to support both OpenMP and MPI wavelength parallelism. Direct execution of `np_sphere` will use all the available system threads. In order to limit thread spawning, the user still needs to set a limit with the `OMP_NUM_THREADS` environment variable.

then run the *FORTRAN* configuration program:

```
~/cluster> ./edfb_clu
```

followed by the *FORTRAN* calculation:

```
~/cluster> ./clu
```

This command sequence will read the default cluster development case, which is a quick calculation of the scattering problem for 2 scales on a cluster made up by 4 spheres. As a result, the two binary files **TEDF** and **TPPOAN** will be written to the current working directory, together with the text file **OCU** containing the results of the calculation.

In a similar way, the *C++* calculation that replicates this process can be executed. If the code was built without parallel optimization compiler flags, the command line to run the test will be just:

```
~/cluster> ./np_cluster
```

In case the code has been built with **OpenMP** enabled, the recommended execution is:

```
~/cluster> OMP_NUM_THREADS=1 ./np_cluster
```

while a build using both **OpenMP** and **MPI** would be tested with:

```
~/cluster> OMP_NUM_THREADS=1 mpirun -n 1 ./np_cluster
```

Setting the **OMP_NUM_THREADS** environment variable to a number *X* different from 1 will cause the code to use *X* **OpenMP** threads (for each **MPI** process, if **MPI** is also used) when parallelizing over wavelength scales.

np_cluster without arguments will look for the same configuration files used by **edfb_clu** and **clu**, namely the **DEDFB** and **DCLU** files stored in **test_data/cluster**, and write **c_**-prefixed output files. The results can be compared by looking in the **OCU** and **c_OCU** files (see § 3.4).

To further test the code, the **test_data/cluster** contains additional testing cases, some of which include the *FORTRAN 66* pre-computed outputs, in text files named **OCU**. The *C++* implementation of the code can directly handle these cases, by running:

```
~/cluster> ./np_cluster PATH_TO_DEDFB_XX PATH_TO_DCLU_XX \  
  
OUTPUT_PATH
```

for a serial implementation, or

```
~/cluster> OMP_NUM_THREADS=X,Y mpirun -n Z ./np_cluster \  
  
PATH_TO_DEDFB_XX PATH_TO_DCLU_XX OUTPUT_PATH
```


where *X*, *Y* and *Z* represent, respectively, the number of **OpenMP** threads dedicated to wavelength parallelism, the number of **OpenMP** threads dedicated to work-sharing tasks, and the number of MPI processes that are intended for use at runtime.⁴

3.2.4 Execution of the inclusion case

Running the **inclusion** case is practically identical to the **cluster** case. The first step is to move to the **inclusion** folder:

```
~/cluster> cd ../inclusion
```

then run the *FORTTRAN* configuration program:

```
~/inclusion> ./edfb_inclu
```

followed by the *FORTTRAN* calculation:

```
~/inclusion> ./inclu
```

This command sequence will read the default inclusion development case, which is a quick calculation of the scattering problem for 2 scales on a spherical particle with an inclusion made up by 2 spheres. As a result, the two binary files **TEDF** and **TPPOAN** will be written to the current working directory, together with the text files **OINCLU**, **scasec**, and **scasecm**, containing the results of the calculation.

Again, the *C++* calculation that replicates this process can be executed for comparison. If the code was built without parallel optimization compiler flags, the command line to run the default test will be just:

```
~/inclusion> ./np_inclusion
```

In case the code has been built with **OpenMP** enabled, the recommended execution is:

```
~/inclusion> OMP_NUM_THREADS=1 ./np_inclusion
```

Similarly to what happens with the execution of **np_sphere** and **np_cluster**, also **np_inclusion** supports wavelength parallelism and custom input / output definition, with the syntax:

```
~/inclusion> OMP_NUM_THREADS=X,Y mpirun -n Z ./np_inclusion \
PATH_TO_DEDFB_XX PATH_TO_DINCLU_XX OUTPUT_PATH
```

where *X*, *Y* and *Z* represent, respectively, the number of **OpenMP** threads dedicated to wavelength parallelism, the number of **OpenMP** threads dedicated to work-sharing tasks, and the number of MPI processes that are intended for use at runtime.

⁴**NP_TMcode** uses two levels of nested **OpenMP** parallelism: the first is for wavelength parallelism (also handled by MPI); the second is for work-sharing. Mixed parallelism is allowed, but the recommendation is to use pure **OpenMP** on workstations and MPI-wavelength + inner level **OpenMP** multi-threading for computing clusters.

3.3 Testing with docker/singularity

Since `NPTM_code` is currently under development, the most straightforward way to obtain an executable version of the code is to download the latest release of the source code and build the program binaries locally. While the process should be fairly straightforward for any Linux based architecture offering the necessary requirements, building on other systems or finding the proper implementation of pre-requisites may not be equally easy. For this reason, the code is also distributed in the form of pre-assembled container images, running under the `docker` and the `singularity` systems.⁵ Refer to the installation instructions on the `docker` and `singularity` web sites to obtain and install one of these, if needed. The free versions are perfectly adequate for testing `NP_TMcode-M10a`. The instructions to recreate local images are given in the `README.md` files in the `docker` and the `singularity` sub-folders of the `containers` directory. Both images include the standard `HDF5` tools, which can be used to inspect the binary output files in `HDF5` format, and compiled implementations using `LAPACK`, `MAGMA`, and `CUDA`.

3.3.1 Docker

A publicly accessible `docker` image can be obtained directly from Docker hub, under the name of `gmulas/np-tmcode-run`.

To test the `NP_TMcode-M10a` in the `docker` image, one can start an interactive shell in the container image instance. This can be achieved either using the `docker` graphical user interface or using the command line, such as e. g.:

```
~/docker> docker run -it gmulas/np-tmcode-run:m10a.00 /bin/bash
```

This will start an instance of the `docker` image, and open an interactive bourne shell within it. Then, one can go into the installation folder of `NP_TMcode` inside the container

```
root@74a5e7e7b79d:~# cd /usr/local/np-tmcode/build
```

and from there proceed as in Sections 3.2.2, 3.2.3, and 3.2.4. Bear in mind that, unless a persistent `docker` volume was created (see `docker` reference documentation to do this), and mounted, on the `docker` image instance, whatever files are created inside the running instance are *lost* when the instance is closed, i. e., in the example above, upon exiting the shell. Whatever one wants to keep should be copied out of the running instance before closing it, e. g. using `docker cp` commands.

⁵Since the `docker` and the `singularity` images are large files, they are not included in the `NPTM_code` source distribution, due to space constraints, and they must be downloaded individually. This is automatically handled by `docker`, while instructions to work with `singularity` are given in § 3.3.2.

3.3.2 Singularity

To test the NP.TMcode-M10a using the `singularity` image, one can make use of the feature of singularity that automatically mounts the user's home directory in the container image, and directly run the programs in the image. The image can be obtained directly from the Sybase cloud of singularity images, from the distribution site with name:

```
gmulas/np-tmcode-run/np-tmcode-run.sif:m10a.00,
```

which contains an implementation built on NP.TMcode-M10a. Before attempting execution, the image file `np-tmcode-run.sif` can be placed in the `containers/singularity` folder of the `np_tmcode` project. The command line to run the image would then be:

```
~/singularity> singularity exec \  
  
COMPLETE_PATH/np-tmcode-run.sif /bin/dash
```

where `COMPLETE_PATH` above is supposed to be the complete path to the singularity image file. This would open a shell within the image. One can therefore proceed as in Sections 3.2.2, 3.2.3, and 3.2.4, just running the executable files via the singularity image file, instead of directly from the host machine. The only caveat is that the *FORTRAN* binaries expect to read their input data from a `test.data` folder located two levels above the singularity execution directory.

Alternatively, and more elegantly, instead of opening a shell within the singularity image, one can directly call specific commands with the simple syntax:

```
~/singularity> singularity run \  
  
COMPLETE_PATH/np-tmcode-run.sif NP_COMMAND
```

where `NP_COMMAND` is any of:

```
edfb_clu, clu, edfb_inclu, inclu, edfb_sph, sph, np_sphere_legacy_PARALLEL,  
np_sphere (defaulting to np_sphere_legacy_mpi), np_trapping, np.CODE  
(defaulting to np.CODE_lapack_mpi), np.CODE_METHOD_PARALLEL, test_cluster_outputs,  
test_inclusion_outputs, test_sphere_outputs, test_ParticleDescriptor, test_TEDF,  
test TTMS, model_maker.py, pycompare.py, pydynrange.py, pytiming.py,
```

where:

`CODE` is one of `cluster`, `inclusion`

`METHOD` is one of `magma`, `cublas`, `lapack`, `legacy`

`PARALLEL` is one of `mpi`, `serial`.

Versions with `METHOD=magma` use the Magma library for matrix inversion, whereas versions with `METHOD=cublas` use the CuBLAS library, versions with `METHOD=lapack` use the LAPACK library (one of the many available implementations), and versions with `METHOD=legacy` use the builtin legacy implementation of the LU factorization.

Versions with `PARALLEL=mpi` are compiled with the OpenMPI implementation of MPI and OpenMP support, whereas versions with `PARALLEL=serial` are compiled without MPI and with only OpenMP support (hence they are still parallelized to some extent). Hence, calling for example `np_cluster_magma_mpi` would call the version of the `np_cluster` code compiled to use the Magma library for matrix inversion and both OpenMPI and OpenMP for the parallel implementation. Indeed, one can use the `np_*` codes as drop-in replacements of locally compiled ones, with exactly the same syntax as described in Sections 3.2.2, 3.2.3, and 3.2.4.

The singularity image also includes the executable python scripts `model_maker.py`, `pycompare.py`, `pydynrange.py`, `pytiming.py`, along with python 3.13 and the libraries they require to run, as well as some `test_*` executables used for internal testing, and all the standard HDF5 utilities.

To further simplify the calling syntax of codes embedded in the singularity container, NP_TMcode-M10a.02 a shell script wrapper, located in `scripts/singwrapper`, which can be linked or renamed to the name of the code one wants to run, and then executed as it is. The wrapper expects to find the full path to the singularity image in the environment variable `SIFFILE`, which, if undefined, defaults to `np-tmcode-run.sif` in the directory above the location of the script being called. Therefore, to use the wrapper to call, e. g., `np_cluster_magma_mpi`, one would link or copy `singwrapper` to that name, define the environment variable `SIFFILE` to point to the singularity image file, then just use:

```
~/singularity> OMP_NUM_THREADS=1,X mpirun -n Y np_cluster \  
  
PATH_TO_DEDFB_XX PATH_TO_DCLU_XX OUTPUT_PATH
```

exactly as previously described in Sect. 3.2.3.

3.4 Comparing results

The comparison of results for a realistic case is, in general, not straightforward. In comparing the output of *FORTRAN 66* and *C++*-based calculations, several effects may introduce artifacts that result in more or less significant differences. The output of the code, indeed, includes both unformatted binary files as well as formatted text files. Due to the facts that only the code is able to read its proprietary binary format and that the formatted output is a following step, it makes perfect sense to compare the results saved in formatted files, since they are derived from the binary ones.

However, comparison of formatted text files can still be a hard task, due to the large amount of information included in each file and to the possibility of observing *numeric noise*. This noise arises on values that are negligible with respect to the typical orders of magnitude probed by a given level of approximation. A similar effect may also be observed when executing the same code on different hardware architectures. In order to make the task of comparing the output of the `sphere` and `cluster` calculations between the *FORTRAN* and the *C++* versions easier, NP_TMcode-M9.01 includes an executable *python3* script named `pycompare.py`. The scope of this script is to parse the formatted output files produced by the *FORTRAN* and the *C++* implementations, to check for the consistency of the file structures and to verify the coincidence of significant numeric values. This script is located in a folder named `src/scripts` and it can be invoked from there with the following syntax:

```
~/scripts> ./pycompare.py --ffile PATH_TO_FORTRAN_RESULT \
--cfile PATH_TO_C++_RESULT
```

where the files to be passed as input are those named `OSPH` and `OCU` by the *FORTRAN* code and `c.OSPH` and `c.OCU` by the *C++* code. The script checks in the result files and it returns a result flag of 0 (OS definition of success), in case of consistent results, or some non-zero integer number (OS indication of failure) otherwise. The script also writes a summary of its diagnostics to the standard output, including the number of inconsistencies that were considered noisy values, warning values (i.e. values with a substantial difference but within a given tolerance threshold) and error values (i.e. values disagreeing by an above-threshold significant difference). If needed, the user may produce a detailed `html` log of the comparison by invoking:

```
~/scripts> ./pycompare.py --ffile PATH_TO_FORTRAN_RESULT \
--cfile PATH_TO_C++_RESULT \
--html [=HTML_LOG_NAME]
```

where the part in square brackets is an optional name of the log file (which, if not specified, defaults to `pycompare.html`). Invoking the script without arguments or with the `--help` option will result in the script printing a detailed help screen on terminal and then exit with success code.

3.5 Model construction with `model_maker.py`

The calculation of new custom models with earlier versions of NP_TMcode required the user to manually provide the necessary input to describe the model parameters, using formatted

ASCII files that could be parsed both by the legacy *FORTRAN* code and by the new *C++* implementation. While this feature was fundamental to perform code stability tests, it also required the user to be familiar with the original input data format.

As shown in § 3.2, the general syntax to run NP_TMcode applications requires the specification of three command line arguments:

1. the spherical units description file (usually named DEDFB);
2. a file describing the particle model and the scattering geometry (DSPH, DCLU, or DINCLU, respectively for `np_sphere`, `np_cluster`, and `np_inclusion`);
3. an output folder (which, as of now, must be an existing directory).

While the structure of these files is shortly explained in the `README.md` files provided in the testing data bundles, the task of manually producing new models, even as simple variants of the test cases, can be very complicated, especially for calculations that need to be repeated over a large number of wavelengths for complicated cluster structures.

In order to assist users in the creation of new models, NP_TMcode-M10a.02 includes a new executable python script, named `model_maker.py` and located in the `src/scripts` folder. The purpose of this script is to parse a human-readable YAML description of the model and to translate it into the proper input files used by NP_TMcode at runtime. To take advantage of this tool, the user needs to issue:

```
~/scripts> ./model_make.py CONFIG_FILE
```

where `CONFIG_FILE` is the full path to a valid YAML configuration file. Examples of such configuration files are provided within the test data folders, but, in essence, a valid configuration file needs to contain the following information (organized in `key : value` tuples):

```
system_settings:
```

```
# Limit on host RAM use in Gb (0 for no configuration limit) NOT YET IMPLEMENTED
max_host_ram : 0
# Limit on GPU RAM use in Gb ( 0 for no configuration limit) NOT YET IMPLEMENTED
max_gpu_ram  : 0
```

```
input_settings:
```

```
# Folder to write the code input configuration files
input_folder : "."
# Name of the scatterer description file
spheres_file : "DEDFB"
# Name of the geometry description file
geometry_file: "DCLU"
```

```
output_settings:
```

```

# Folder for the code output storage
output_folder: "."
# Name of the main output file NOT YET IMPLEMENTED
output_name : "c_OCLU"
# Requested output formats NOT YET IMPLEMENTED
formats      : [ "LEGACY", "HDF5" ]
# 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   : 4
# Number of sphere types
n_types     : 4
# 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    : [ 1, 1, 1, 1 ]
# Spherical monomer radii in m (one size for each type)
radii       : [ 5.0e-08, 4.0e-08, 7.5e-08, 3.0e-08 ]
# Layer fractional radii (one per layer in each type)
rad_frac    : [ [ 1.0 ], [ 1.0 ], [ 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 ], [ 1 ], [ 1 ] ]

material_settings:
# Type of dielectric constants: -1 = manually defined, 0 = from data file
diel_flag   : 0
# External medium dielectric constant
extern_diel : 2.3104e+00
# Dielectric constant files folder
dielec_path : ".././ref_data"
# List of dielectric constant files (used if diel_flag = 0)
dielec_file : [ "eps_draine_Si.csv" ]
# Dielectric constant files format (same for all files) NOT YET IMPLEMENTED
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 : "INTERPOLATE"
# 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 : 1.0e-07
# Last scale to be used
scale_end : 1.0e-06
# Calculation step (overridden if 'match_mode' is GRID)
scale_step : 5.0e-09
# Peak Omega
wp : 2.99792e+08
# Peak scale
xip : 1.0e+00
# Define scale explicitly (0) or in equal steps (1)
step_flag : 0
# Type of scaling variable (XI | WAVELENGTH)
scale_name : "WAVELENGTH"

geometry_settings:
# Maximum internal field expansion
li : 8
# Maximum external field expansion (not used by SPHERE)
le : 8
# 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

```



```

# Starting incidence azimuth angle
in_th_start : 79.0
# Incidence azimuth angle incremental step
in_th_step  : 10.0
# Ending incidence azimuth angle
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 X coordinates of sphere centers (one per sphere or empty for random)
x_coords    : [
    0.00e+00,
    0.00e+00,
    1.18882e-07,
    -5.656855e-08,
]
# Vector of Y coordinates of sphere centers (one per sphere or empty for random)
y_coords    : [
    3.00e-08,
    3.00e-08,
    3.00e-08,
    -2.656855e-08
]
# Vector of Z coordinates of sphere centers (one per sphere or empty for random)
z_coords    : [
    0.00e+00,

```

```
9.00e-08,  
3.8627e-08,  
0.00e+00  
]
```

The above example sets up for the calculation of the scattering problem from a cluster of four spheres, using the tabulated optical constants of astronomical silicates (as given by Draine & Lee (1984)), interpolating them for a wavelength range going from $0.1\ \mu\text{m}$ to $1.0\ \mu\text{m}$ in steps of $5\ \text{nm}$.

A set of example **YAML** configuration files illustrating possible variations, with respect to the example above, is included in the test data folder. In particular, `NP-TMcode-M10a.02` provides the configuration files to derive all the development test cases. For more detailed information on how to use the `mode_maker.py` script, the interested reader is referred to the script inline documentation and the script help mode, activated with:

```
~/scripts> ./model_make.py --help
```

Bibliography

- Borghese, F., Denti, P., & Saija, R. 2007, *Scattering from Model Nonspherical Particles* (Berlin: Springer)
- Draine, B. T. & Flatau, P. J. 1994, *J. Opt. Soc. Am. A*, 11, 1491
- Draine, B. T. & Lee, H. M. 1984, *The Astrophysical Journal*, 285, 89
- Iatì, M. A., Giusto, A., Saija, R., et al. 2004, *The Astrophysical Journal*, 615, 286
- Palik, E. D. 1991, "Handbook of optical constants of solids II" (Elsevier)