Nano-particle Transition Matrix code Release Notes

Report for M8.02

G. La Mura, G. Mulas

June 2024

Contents

Scope of the document

This document is provided along with the M8 release of the NP-TM code 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.

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, also included in the distribution. The introductory discussion presented in Chapter 1 is similar to the one given in the previous release. The contents of Chapters 2 and 3 assume that the reader is aware of the discussion presented in the release notes of M7.00.

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 single 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* (TM) 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 act as a mathematical operator connecting the properties of the incident and scattered field.

The TM 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 TM 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 TM.

Conversely, DDA calculations need to be entirely solved for any 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 TM 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 TM 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. This 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++ (completed in M7)

2. **implementation of parallel algorithms** (addressed in this release)

3. deployment of general radiation/particle interaction solver

These three stages are associated with an equal number of Key Performance Indicators (KPI) which consist in code releases with enhanced computational abilities. The current project stage provides a first parallel implementation of the calculation, built on the basis of the $C++$ ported algorithms, taking advantage of the profiling analysis carried out in the previous release to reduce the execution time of the most intensive calculation steps.

1.3   Project status

We refer to the current release of the project as `NP_TMcode-M8.02`, since it includes updates on the documentation and the license of the code implementation that addresses the targets of the project Milestone 8 (namely, a first parallel implementation of the solution of the `cluster` problem and an analysis of the performance improvement with respect to the original code). 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-M7.0`:

- the possibility to compile the code as either a serial or a parallel software application

- the option to link against optimized linear algebra external libraries (to speed up calculations) or to use legacy algorithm, in case optimized libraries are not available

- the possibility to offload the matrix inversion calculation to GPUs, if available

- the control of code execution by means of environment variables

- a multi-threaded / multi-node parallel implementation designed to scale on large computational facilities.

The first three features give users the chance to build the code both on personal workstations and on shared computing systems with different hardware characteristics. The last two features, instead, are intended to give users a certain degree of control on the complexity of the calculation and on the amount of required resources at runtime, enhancing the user's ability to execute calculations in shared computing environments.

Chapter 2

Release description

2.1 Parallel implementation

`NP_TMcode-M8.02` is the second 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 evaluate the performance improvement achieved over the original *FORTRAN 66* code, through the porting of its algorithms to *C++* and their subsequent implementation in parallel architectures.

The profiling analysis carried out during the completion of M7 has led to the identification of two promising parallelization strategies:

1. The simultaneous solution of the problem at different wavelengths.

2. The re-implementation of standard linear algebraic tasks (such as matrix and vector operations) within optimized parallel libraries.

Due to the problem dependence on radiation wavelength, the calculation of scattering in different wavelengths is nearly independent, except for a few configuration steps, and it can be carried out as an embarrassingly parallel multi-thread or multi-process task. `NP_TMcode-M8.02` implements this feature using a combination of `OpenMP` thread management and `MPI` process

---

[1] The first official release and its documentation is available at `https://www.ict.inaf.it/gitlab/giacomo.mulas/np_tmcode/-/releases/NP_TMcode-M7.00`.

handling that can be optionally activated, by means of proper compilation flags, and subsequently affected through the use of environment variables and execution configurations.

For what concerns the linear algebraic tasks, `NP_TMcode-M8.02` offers the possibility to switch among the internal legacy algorithms and external optimized libraries, such as the *Linear Algebra Package* (`LAPACK`). Linking with `LAPACK` is possible in different flavours, using either serial implementations, like e.g. `LAPACKe`, or parallel oriented ones, such as `OpenBLAS` and Intel `MKL`. More details on how to configure and use the code to access these possibilities are given in Chapter 3 of this document.

2.2   New release features

Compared with the previous release, `NP_TMcode-M8.02` has some fundamental improvements. These mostly concern the code management of resources at runtime and they do not require large changes in the way the user is expected to work with the code. Many of the newly implemented features are optional, meaning that the code can still be compiled and executed in the old-fashioned serial implementation by systems that do not have parallel compilation and execution capabilities. To activate the new features, users are required to go through a more advanced compilation step, defining a set of compiler flags, and then to configure the code execution using system environment variables. The three main features that the current release adds over the previous one are:

1. Parallel simultaneous execution of the calculation at different wavelengths.

2. Optional offload of matrix inversion to GPU devices.

3. Implementation of virtual output management to write results computed simultaneously.

Except for requiring more details in the compilations and the execution command lines, these features work behind the stage, but their effect is clearly
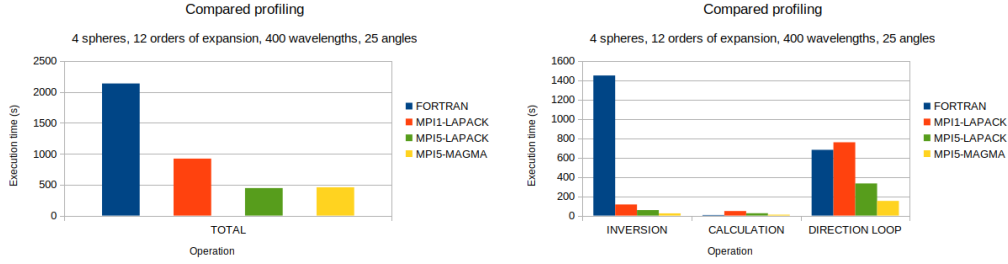
Figure 2.1: Comparison of the benchmark calculation execution times for different code implementations. The left panel shows the total execution time, while the right panel illustrates a breakdown of the main resolution steps (matrix calculation, matrix inversion and direction loop). The legacy *FORTRAN 66* implementation is compared with three configurations of NP_TMcode-M8.02: single process with `LAPACK`, 5 simultaneous processes with `LAPACK`, and 5 simultaneous processes with `MAGMA`-driven GPU offload of matrix inversion.

visible in terms of performance analysis. In order to illustrate the performance gain, a set of new test cases has been added to the cluster calculation examples and a discussion of their execution, with different configurations on the same hardware, is presented in the following section.

2.3 Code performance

The code performance for Milestone 8 has been evaluated through a new benchmark test case (named `case_3` in the test data suite provided for the `cluster` code). This benchmark case simulates the scattering of radiation by a particle composed of one gold sphere, with radius of $0.20\,\mu$m, and three smaller spheres of plastic material, having radii of $0.02\,\mu$m, $0.03\,\mu$m, and $0.05\,\mu$m, respectively. In order to test the widest range of possible code functionalities, the result of the scattering is evaluated for a total of 401 radiation field wavelengths, ranging from $\lambda = 400\,$nm to $800\,$nm, and evaluating the integrated interaction cross-sections, together with 25 differential ones, cor-

Table 2.1: Execution times of the breakdown steps and total calculation times of the configurations illustrated in Fig. 2.1. Note that the total execution times are subject to different types of overheads, such as GPU initialization and finalization stages for the `MAGMA` execution, or output concatenation for the multi-process configurations, resulting in total times that are larger than the sum of the single breakdown times.

CONFIG.	CALC. TIME	INV. TIME	DIR. LOOP	TOTAL
LEGACY	5.75 s	1450.01 s	680.33 s	2136.09 s
MPI1-LAPACK	46.92 s	115.32 s	758.62 s	921.13 s
MPI5-LAPACK	24.1 s	56.82 s	332.61 s	444.80 s
MPI5-MAGMA	9.38 s	23.06 s	151.72 s	457.42 s

responding to an equal number of required scattering angles.[2]

The performance of the code has been evaluated by running the calculation of the benchmark case on an ASUS Zenbook laptop computer, with 32 Gb of RAM, CPU 13th Gen Intel(R) Core$^{\text{TM}}$ i9-13900H x 20, featuring a NVIDIA GeForce RTX 4060 Laptop GPU with 8 Gb of dedicated memory. The results of the benchmark test are illustrated in Fig. 2.1 and reported in detail in Table 2.1.

As it is illustrated in Fig. 2.1, the calculation time can be broken in three main stages: a matrix calculation stage, a matrix inversion stage and a loop on the requested differential scattering cross-sections. The benchmark calculation shows that, using the same hardware resources, the `NP_TMcode-M8.02` implementation improves over the performance of the legacy code, though with different efficiency in the various steps. Forcing a serial execution with a single MPI process, the main improvement that we observe is the substantial reduction of the matrix inversion step, obtained by replacing the legacy code

---

[2]The output of the benchmark case in the current default format takes 2.7 Gb of space and it is therefore not included in the distribution. To test for consistency of the results with the legacy *FORTRAN 66* implementation, however, we include in the distribution the output produced for smaller datasets, namely including a a calculation executed on 33 different wavelengths and another executed on 65.

algorithm with calls to specialized `LAPACK` functions. In the other steps, the *FORTRAN 66* implementation is slightly faster than the *C++* one, due to the necessary overheads that *C++* needs to work with in order to dynamically allocate and release the memory resources, in place of using a predetermined memory stack, defined at compiling time, as in *FORTRAN*.

A much clearer advantage stems from the exploitation of multi-process capabilities, thus enabling the simultaneous solution of different wavelengths at once. In this case, the most convenient parallel strategy appears to be the execution of simultaneous processes in a MPI implementation. The scaling that can be achieved is not linear, due to system overheads in performing low level tasks, such as device initialization and finalization (for GPU offloaded calculations), or data broadcasting and result concatenation for pure MPI operations. The use of multi-threaded approaches to perform embarrassingly parallel operations on the wavelength calculation is under-performing, with respect to MPI managed calculations, while it has a substantial impact on operations that require shared memory access, such as matrix inversions. In this specific case, the excellent multi-threading capabilities of GPU devices lead to substantial overall improvement among the *C++*based configurations. A detailed investigation of the calculation overheads and of the performance scalability on more powerful hardware implementations will be covered by the targets of the next development milestone.

Chapter 3

Instructions for testing

3.1   Set up operations

Testing `NP_TMcode-M8.02` 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. This section deals with testing the code release on a local machine. Detailed instructions on how to use pre-built images, instead, are given in § 3.4, later on.

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.

3.1.1   Building the code

To build the code, the user needs a set of compilers and some libraries. The recommended set up includes an up to date installation of the *GNU Compiler Collection* (`gcc`), of the *GNU* `make` builder, a *FORTRAN* compiler (again, the recommended option is to rely on *GNU*'s `gfortran`) and the *doxygen* document manager. An optional dependency is a working LaTeXdistribution with recommended package set-up, in order to build the full PDF documentation. If the aforementioned system requirements are met, building the code

just requires to go in the `src` folder:

```
~/np_tmcode> cd src
```

and then invoking `make`.[1] The compilation can include several optional compiler flag, which enable different parallel implementation and optimization features. The list of valid compiler flags is:

FC=FORTRAN_COMPILER name of the compiler used to build the *FORTRAN* legacy code

CXX=CPP_COMPILER name of the compiler used to build the *C++* code

USE_LAPACK=1 compiler flag to enable `LAPACK`

USE_ILP64=1 compiler flag to enable 64-bit `LAPACK` indices[2]

USE_MAGMA=1 compiler flag to enable `MAGMA` handling of GPU offload

USE_MKL=1 compiler flag to enable `MKL` implementation of `LAPACK`

USE_MPI=1 compiler flag to enable MPI parallelism on wavelengths (requires setting `CXX=mpicxx`)

USE_NVTX=1 compiler flag to enable NVIDIA profiling tools (requires NVIDIA Tools to be available on the system)

USE_OPENMP=1 compiler flag to enable OpenMP multi-threading

For example, the recommended way to build `NP_TMcode-M8.02` on a multi-core 64-bit machine with available GPU, MPI and MAGMA libraries would be:

---

[1]In 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.

[2]The use of the correct type for vector indexing is fundamental when linking with external libraries. Some `LAPACK` implementations use 32-bit integer type for indices, while others use 64-bit. Therefore, when building the code, attention must be paid to use the correct indexing convention. If the chosen index type does not match the one used by the available libraries, the compilation may still run, but the code execution will lead to errors and abrupt termination.

```
~/src> USE_OPENMP=1 USE_MPI=1 USE_LAPACK=1 USE_ILP64=1 \

    USE_MAGMA=1 FC=gfortran CXX=mpicxx make -j
```

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, there is the additional option to build the code inline documentation by issuing:

```
~/src> make docs
```

which will generate a folder named `doc/build` under the 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):

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

3.2 Execution of the `sphere` case

Once the build process has been completed, the code is ready to be run on the available test cases. The configuration files and the expected *FORTRAN* output files 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:

```
~/src> cd ../build/sphere
```

then run the *FORTRAN* configuration program:

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

The `edfb_sph` program looks for the problem configuration data in a file named `DEDFB`[3] 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> ./np_sphere
```

The `np_sphere` program adopts the default behaviour of looking for the same input data as the *FORTRAN* code and writing the same type of output files, but appending a `c_` prefix to its output. Optionally, it can be run as:

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

to let it get input and write output other than the default behaviour.

After the execution of the *FORTRAN* and the *C++* versions, the final outcome can be compared by checking the contents of the `OSPH` and the `c_OSPH` files (see § 3.5 for suggestions on how to compare files).

3.3   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
```

then run the *FORTRAN* configuration program:

---

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

```
~/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 `OCLU` 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 the `USE_OPENMP=1` flag, 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
```

`np_cluster` 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 `OCLU` and `c_OCLU` files (see § 3.5).

To further test the code, the `test_data/cluster` contains further testing cases, some of which include the *FORTRAN 66* pre-computed outputs, in text files named `OCLU`. 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=1,X mpirun -n Y ./np_cluster \

            PATH_TO_DEDFB_XX PATH_TO_DCLU_XX OUTPUT_PATH
```

where `X` and `Y` represent, respectively, the number of OpenMP threads and MPI processes that are intended for use at runtime.[4]

3.4 Testing with `docker/singularity`

The project release contains a `containers` directory, which in turn contains `docker` and `singularity` subdirectories. These subdirectories contain configuration files which can produce working container images using either container system, which is assumed to be installed on the testing machine. 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-M8.02`. The instructions to recreate local images are given in the `README.md` files in the respective subdirectories. Moreover, a publicly accessible docker image can also be obtained directly from Docker hub, under the name of `gmulas/np-tmcode-run`, and a pre-built singularity image file `np-tmcode-run.sif` is available under the `singularity` subdirectory. Both images include also the standard `HDF5` tools, which can be used to inspect the binary output files in `HDF5` format.

To test the `NP_TMcode-M8.02` 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:M8 /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

---

[4]The recommended way to use the current release of the code is to always set the `OMP_NUM_THREADS` environment variable in such a way that the first layer of OpenMP threading is configured to 1 thread. In case of use of multiple threads at the first threading layer, OpenMP will start multiple threads on the wavelength range, which may result in resource consumption and performance degradation.

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

and from there proceed as in Sections 3.2,3.3, and 3.5. 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.

To test the NP_TMcode-M8.02 in 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 command line to run the image would then be:

```
~/singularity> singularity exec \

               COMPLETE_PATH/np-tmcode-run.sif /bin/bash
```

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, 3.3, and 3.5, 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.

3.5   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-M8.02 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 `OCLU` by the *FORTRAN* code and c_`OSPH` and c_`OCLU` 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]
```

19

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.

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