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 Release Notes

Report for M7.00

G. La Mura, G. Mulas

January 2024

Contents

Scope of the document 2

1 Aim of the project **3**

1.1 Introduction 3

1.2 Project break-down 5

1.3 Project status 5

2 Release description **7**

2.1 Code migration 7

2.2 New code features 8

2.3 Code performance 9

3 Instructions for testing **12**

3.1 Set up operations 12

3.1.1 Building the code 12

3.2 Execution of the `sphere` case 14

3.3 Execution of the `cluster` case 15

3.4 Testing with `docker/singularity` 16

3.5 Comparing results 17

Scope of the document

This document is provided along with the 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 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.

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 and relative orientations of incident fields and the complex scattering particle. E. g. 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++** (currently being finalised)
2. implementation of parallel algorithms
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 first project stage, addressed by the current release, aims at migrating the original code to a modern language. This allows to set up an execution framework that can be configured by modern compilers and make use of hardware resource and speed optimization features, such as dynamic memory management.

The first stage also provides the necessary framework to estimate runtime system requirements and to perform a detailed profiling of the calculation, in order to lay down the most convenient strategy for the subsequent parallelization stage, and a fundamental inline documentation, handled by *doxygen*, to describe in further detail every code component.

1.3 Project status

We refer to the current release of the project as `NP_TMcode-M7.0`, since it is the first (pre-evaluation) code release addressing the targets of the project Milestone 7 (namely a consistent re-implementation of the `sphere` and `cluster` calculation cases, able to reproduce the legacy results for a set of development cases with a profiling analysis).[1] This code has been designed to emulate as closely as possible the work-flow of the original *FORTRAN*

[1]The code included in this release performs a full calculation reproducing the original `sphere` and `cluster` programs and includes a development version of the `trapping` program. The code is built with integration of profiling tools, but a full profiling analysis is expected by the end of February 2024.

version to allow for direct performance comparison and easy identification of calculation bottle-necks.

In addition to reproducing the legacy results of the development test cases, the *C++* implementation already offers some advantages with respect to the original code. The most important ones are:

- the possibility to use command line arguments to choose input and output paths
- the use of classes in place of common data blocks
- runtime evaluation of memory requirements, and consequent dynamical allocation of variables, whose sizes are not hard-coded as in the original code
- optional output to standard binary file formats (`HDF5`), enabling cross-platform portability and comparison of calculations

The first three features allow users to run the calculation of different scattering problems without having to adjust the source code. The last feature, instead, adds the possibility to redirect part of the code output towards data structures that can be inspected and possibly converted by external software tools. This also makes it possible to analyse results in binary formats on a different platform from the one in which the calculation was performed, regardless of endianness, with the only requirement that the `HDF5` library is available on both.

Chapter 2

Release description

2.1 Code migration

`NP_TMcode-M7.0` is the first release of the *Nano-particle Transition Matrix Project* code, 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 verify the reproducibility of the results obtained by the original *FORTRAN 66* code in a new version, implemented in a more recent programming language.

The language chosen for the new implementation is *C++*, due to its optimal balance between low-level hardware control, execution speed, language diffusion and dynamic resource management at runtime. The goal of this first release is to provide a complete suite of tests that can be used to run both the *FORTRAN* and *C++* versions of the code, to verify the consistency of the results and to assess the overall code performance.

This release includes the *C++* source code and its inline documentation, together with a collection of essential test cases, made up by the required input and the *FORTRAN* code that has been explicitly configured to run them. In order to execute this test suite, the user needs a set of tools and libraries, required to build the binary executable files, which have been also documented in mark-down formatted `README` files. Given the typical limitations of source code distributions, which require tools to build an executable version of the code that, so far, have only been tested under some

Linux and macOS system architectures, the release also includes a set of preconfigured test suites, distributed in the form of `docker`[1] and `singularity`[2] container images, containing precompiled binary executables along with a minimal Linux system in which they can be run.

2.2 New code features

The goal of the `NP_TMcode-M7.0` code release is to set up a working implementation of the scattering problem solution for the cases of radiation impinging on a single sphere (with arbitrary layer structure) and on a cluster of spheres in *C++*, obtaining consistent results with respect to the original programs. The achievement of this goal can be verified by executing the original *FORTRAN* code and the new *C++* version on the same input files. Details on how to perform these operations are laid down in Chapter 3. In this section, instead, we discuss some of the features that were added to the code in the migration stage.

Indeed, due to the limited scalability of *FORTRAN 66*, with respect to *C++*, the original code has its input and output configuration defined within the source (i. e. the names of input and output files are hard-coded, as well as the maximum orders of the truncated expansions). As a consequence, any possible change in the input parameters, requires consequent changes in the *FORTRAN* source files, which subsequently need to be compiled again, before execution. In *C++*, this limitation can be overcome by the combination of command-line arguments, which allow to choose different input and output configurations, and dynamic memory management, which, instead, gives the code the opportunity to calculate the amount of necessary system resources (chiefly the memory space that needs to be used) after parsing the input, thus relieving the user from the burden of computing the hardware requirements depending on the desired degree of complexity and accuracy.

In addition to the possibility of switching between different input and output paths and to change the calculation parameters without having to modify the source code, `NP_TMcode-M7.0` introduces a wrapper class that

[1]see www.docker.com

[2]see sylabs.io/singularity

allows to perform I/O operations on binary files using the HDF5 format. This feature, which is currently included as an optional possibility, provides the opportunity to read and write calculation results in a standard format that may be used by external tools to perform tasks such as, for instance, plotting and conversion, with complete portability across any platform supporting `HDF5`. These represent a fundamental aspect, in order to obtain advanced products like publication quality tables and plots, and the platforms for post-production analysis are completely decoupled from the platforms on which the codes are run for production.

2.3 Code performance

When compiled with the same optimisation options (e. g. `-O3` for both *FORTRAN* and *C++*), corresponding codes consistently take the same time to run, showing that the mere porting to *C++* did not, in itself, imply any significant performance penalty. The minimal difference can be entirely ascribed to the added functionalities (dynamic memory handling, terminal output, and additional `HDF5` binary output).

As an example, we here include excerpts of the profiling we ran for a calculation with a cluster of 48 spheres, both for the *FORTRAN* and *C++* versions of the code. The execution of the *FORTRAN* version yielded the following profiling report:

```
Flat profile:


Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 96.35    107.18   107.18      181     0.59     0.59  lucin_
  2.89    110.40     3.22 26133685     0.00     0.00  ztm_
  0.47    110.92     0.52 40868352     0.00     0.00  r3jjr_
  0.07    111.00     0.08 91953792     0.00     0.00  cgev_
  0.05    111.06     0.06      181     0.00     0.02  cms_
  0.04    111.11     0.05   213037     0.00     0.00  rbf_
  0.04    111.16     0.05                             _init
  0.03    111.19     0.03  1112064     0.00     0.00  ghit_
  0.03    111.22     0.03        1     0.03   111.19  MAIN__
  0.01    111.23     0.01   213218     0.00     0.00  msta2_
  0.01    111.24     0.01      181     0.00     0.00  hjv_
  ... CALLS TO FUNCTIONS BELOW SAMPLING THRESHOLD ...
  0.00    111.24     0.00        1     0.00     0.00  wmamp_
```

Conversely, the *C++* implementation resulted in:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 97.32    120.04   120.04      181     0.66     0.66  lucin()
  1.39    121.75     1.71      183     0.01     0.01  ztm()
  0.51    122.38     0.63 27245568     0.00     0.00  ghit()
  0.29    122.74     0.36 40868352     0.00     0.00  r3jjr()
  0.13    122.90     0.16 91953792     0.00     0.00  cgev()
  0.13    123.06     0.16      181     0.00     0.01  cms()
  0.10    123.18     0.12        1     0.12   123.29  cluster()
  0.04    123.23     0.05                             _init
  0.02    123.26     0.03   213037     0.00     0.00  rbf()
  0.02    123.28     0.02   213218     0.00     0.00  msta2()
  0.02    123.30     0.02   204349     0.00     0.00  rnf()
  0.01    123.31     0.01  1778584     0.00     0.00  dconjg()
  0.01    123.32     0.01   213218     0.00     0.00  msta1()
  0.01    123.33     0.01      181     0.00     0.00  apc()
  0.01    123.34     0.01      181     0.00     0.00  scr2()
  ... CALLS TO FUNCTIONS BELOW SAMPLING THRESHOLD ...
  0.00    123.34     0.00        1     0.00     0.00  ~C1_AddOns()
```

It can be easily verified that the two versions have comparable execution times, with *C++* taking slightly longer to run (123.34s vs. the 111.24s of the *FORTRAN* implementation), due to the additional overheads of dynamic memory management and more verbose terminal output. Apart from this, the two versions have very similar function call footprints, except for the fastest calls and some compilation related subtleties.

As it stands out clearly in this profiling analysis, the largest computational effort (taking more than 95% of the execution time) is the inversion of the Transition Matrix, performed by the `lucin` function. This will therefore be the natural target of parallel code optimization.

Chapter 3

Instructions for testing

3.1 Set up operations

Testing `NP_TMcode-M7.0` 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
```

if necessary to edit (with any text editor) the `make.inc` file (or to override its settings by defining corresponding environment variables), which sets the default compilers, compiler options, paths to the `HDF5` include files and libraries, and then to issue the `make` command:

```
~/src> make
```

If multiple cores are available, a quicker build execution can be obtained by issuing

```
~/src> 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`[1] 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
```

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

```
~/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 by issuing:

```
~/cluster> ./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 two additional cluster configuration files, composed by 24 and 48 spherical units each. The *FORTRAN* code is not able to handle these files, because it would require modifying the source code to read them. Such modifications where tested on the development machine and the results were bundled in the release as the files named OCLU_24 and OCLU_48 in the test_data/cluster folder. The *C++* implementation, on the contrary, can directly handle these, by running:

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

           OUTPUT_PATH
```

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-M7.0. 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-M7.0 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:M7.00 /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 3.2,3.3, 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-M7.0` 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 executables in the image as, e. g.

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

where `COMPLETE_PATH` above is supposed to be the complete path to the singularity image file. 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-M7.0` 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]
```

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