User Manual for the Discrete Dipole Approximation Code ADDA v. 0.79

Maxim A. Yurkin

Institute of Chemical Kinetics and Combustion, Siberian Branch of the Russian Academy of Sciences, Institutskaya 3, Novosibirsk, 630090, Russia, tel: +7-383-333-3240, fax: +7-383-334-2350

Alfons G. Hoekstra

Faculty of Science, Section Computational Science, of the University of Amsterdam, Kruislaan 403, 1098 SJ, Amsterdam, The Netherlands, tel: +31-20-525-7462, fax: +31-20-525-7490

email: adda@science.uva.nl
adda-discuss@googlegroups.com

last revised: May 29, 2009

Abstract

This manual describes using of the **ADDA** code, which simulates elastic light scattering from finite 3D objects of arbitrary shape and composition in vacuum or non-absorbing homogenous media. **ADDA** allows execution on a multiprocessor system, using MPI (message passing interface), parallelizing a *single* DDA calculation. Hence the size parameter of the scatterer, which can be accurately simulated, is limited only by the available size of the supercomputer. However, if the refractive index is large compared to 1, computational requirements significantly increase.

ADDA can be installed on its own, or linked with the FFTW 3 (fastest Fourier transform in the West) package. The latter is generally significantly faster than the built-in FFT, however needs separate installation of the package.

ADDA is written in C and is highly portable. It supports a variety of predefined particle geometries (ellipsoid, rectangular solids, coated spheres, red blood cells, etc.) and allows importing of arbitrary particle geometry from a file. **ADDA** automatically calculates extinction and absorption cross sections and the complete Mueller matrix for one scattering plane. The particle may be rotated relative to the incident wave, or results may be orientation averaged. This manual explains how to perform electromagnetic scattering calculations using **ADDA**. CPU and memory usage are discussed.

This manual can be cited as:
M.A. Yurkin and A.G. Hoekstra "User manual for the discrete dipole approximation code ADDA v.0.79", http://a-dda.googlecode.com/svn/tags/rel_0_79/doc/manual.pdf (2009).

Contents

1 Introduction

ADDA is a C software package to calculate scattering and absorption of electromagnetic waves by particles of arbitrary geometry using the discrete dipole approximation (DDA). In this approximation the volume of the scatterer is divided into small cubical subvolumes ("dipoles"), interaction of which is considered approximately based on the integral equation for the electric field [1]. Initially DDA (sometimes referred to as the "coupled dipole approximation") was proposed by Purcell and Pennypacker [2] by replacing the scatterer by a set of point dipoles (hence the name of the technique). DDA theory (considering point dipoles) was reviewed and developed further by Draine and coworkers [3-6]. Derivation of DDA based on the integral equation for the electric field was apparently first performed by Goedecke and O'Brien [7] and further developed by others (e.g. [8-11]). It is important to note that the final equations are essentially the same (small differences are discussed in §11). Derivations based on the integral equations give more mathematical insight into the approximation, while the model of point dipoles is physically more clear. An extensive review of DDA, including both theoretical and computational aspects, was recently performed by the authors [12].

ADDA is a C implementation of the DDA developed by the authors. The development was conducted by Hoekstra and coworkers [13-16] for more than 10 years in University of Amsterdam. From the very beginning the code was intended to run on a multiprocessor system or a multicore processor (parallelizing a *single* DDA simulation). See §H for a brief history of early stage of **ADDA** development. The code was significantly rewritten and improved by Yurkin [17], during his work on his thesis in University of Amsterdam. Since then the authors has been further developing the code. Originally the full name of the code was "Amsterdam DDA"; however the first word has been officially abbreviated to reflect the international nature of **ADDA** development. Now the first A in the name may hold a variety of meanings: Amsterdam, Akademgorodok (to reflect current affiliation of one of the authors), a (to once become *the*), advanced (what the authors hope to achieve), etc. However, the official name is just **ADDA**.

ADDA is intended to be a versatile tool, suitable for a wide variety of applications ranging from interstellar dust and atmospheric aerosols to biological particles; its applicability is limited only by available computer resources (§6). As provided, **ADDA** should be usable for many applications without modification, but the program is written in a modular form, so that modifications, if required, should be fairly straightforward.[1]

The authors make this code openly available to others, in the hope that it will prove a useful tool. We ask only that:

- If you publish results obtained using **ADDA**, you should acknowledge the source of the code. A general reference [17] is recommended for that.
- If you discover any errors in the code or documentation, please submit it to the **ADDA** issue tracker.[2]
- You comply with the "copyleft" agreement (more formally, the GNU General Public License v.3[3]) of the Free Software Foundation: you may copy, distribute, and/or modify the software identified as coming under this agreement. If you distribute copies of this software, you must give the recipients all the rights which you have. See the file `doc/copyleft` distributed with the **ADDA** software.

[1] However, in some parts modularity was sacrificed for the sake of performance. E.g. iterative solvers (§13.1) are implemented not to perform any unnecessary operations (which usually happens when using standard libraries).
[2] http://code.google.com/p/a-dda/issues
[3] http://www.gnu.org/copyleft/gpl.html

We strongly encourage you to identifying yourself as a user of **ADDA** by subscribing to `adda-announce` Google group;[4] this will enable the authors to notify you of any bugs, corrections, or improvements in **ADDA**. If you have a question about **ADDA**, which you think is common, please look into the `doc/faq` file before searching for the answer in this manual. If neither of these helps, direct your questions to `adda-discuss` Google group.[5] The archive of this group is available,[6] so you may try to find an answer to your question before posting it. However, this also means that you should *not* include (potentially) confidential information in your message; in that case contact one of the authors directly. We also encourage users interested in different aspects of **ADDA** usage to join this group to participate in discussions initiated by other people.[7]

This manual assumes that you have already obtained the C source code for **ADDA** (see §3 for instructions). In §2 we describe the principal changes between current and previous releases. The succeeding sections contain instructions for:

- compiling and linking the code (§4);
- running a sample simulation (§5);
- defining a scatterer (§8) and its orientation (§9);
- specifying the type and propagation direction of the incident beam (§10);
- specifying the DDA formulation (§11);
- specifying what scattering quantities should be calculated (§12);
- understanding the computational aspects (§13) and timing of the code (§14);
- understanding the command line options (§A) and formats of input (§B) and output (§C) files.
- modifying the source code for added functionality (§F).

Everywhere in this manual, as well as in input and output files, it is assumed that all angles are in degrees (unless explicitly stated differently). The unit of length is assumed μm, however it is clear that it can be any other unit, if all the dimensional values are scaled accordingly.

2 What's New

The most important changes between **ADDA** 0.79 and 0.78.2 are the following:

- Added using SLURM job ID, when available (§C.3).
- Saving of granule coordinates to a file was implemented (§8.5).
- The code was tested for 64 bit, implementing a number of corrections/improvements.
- New shape `axisymmetric` was added. The internals of shape generation routines were adjusted so that shape can now define the absolute size of the particle (§8.4).
- A new option "`-scat fin`" was implemented, which uses finite dipole correction for calculation of C_{abs} and C_{ext} (§11.3).
- Breakdown detection of iterative solvers was improved (§13.1).
- `-prognose` command line option was replaced by `-prognosis` (correct spelling, §7). The former is still operational but marked as deprecated.
- The code was transformed to C99 standard without using its parts, which are not yet widely supported by compilers (such as inline and complex). Explicit support of this standard is required during compilation (§4.1). Some style changes were done, and some work in this direction is still due.

[4] To do this send an e-mail to `adda-announce-subscribe@googlegroups.com`
[5] Just send an e-mail to `adda-discuss@googlegroups.com`
[6] `http://groups.google.com/group/adda-discuss/topics`
[7] To do this send an e-mail to `adda-discuss-subscribe@googlegroups.com`

- Affiliation of one of the author (Maxim Yurkin) has changed to Institute of Chemical Kinetics & Combustion (Novosibirsk, Russia). Full name of the package was changed from 'Amsterdam DDA' to 'ADDA'. This does not change the global strategy of **ADDA** development.
- Code repository has been moved to Google Code.[8] Google groups `adda-announce`,[4] `adda-discuss`,[6] and `adda-develop` were created to facilitate interaction with users. This replaces the previous mailing list
- Copyright was updated to GPL 3.[3]
- To-do list was replaced by issue tracker.[2]
- Additions to the manual: short history of earlier **ADDA** development (§H), description how to add new beam type (§F.2) and new command line options (§F.3), a small summary of the comparison of the DDA with the FDTD (§E.3), discussion of difference between two ways to calculate C_{sca}, possibilities to improve the accuracy of these calculations (§11.3), and connection between differential cross section and the Mueller matrix (§12.1).
- Information on how to cite the manual was added to its first page. It includes a direct link to the particular version of the manual on Google code.
- Manual license is now explicitly set to Creative Commons Attribution 3.0 license.
- Several minor bugs were fixed.

The full history of **ADDA** releases and differences can be found in `doc/history`.

3 Obtaining the Source Code

ADDA is a free software (§1). The recent release can be downloaded from:
`http://code.google.com/p/a-dda/downloads`
If you want to use the latest features or track the development progress, you may download the recent source directly from the **ADDA** subversion repository using web browser[8] or any subversion client.[9] Please mind, however, that the latest pre-release version may be unstable. If you decide to try it and discover any problems, comment on this in the issue tracker.[2]

The main package contains the following:

`doc/` – documentation
- `copyleft` – GNU General Public License
- `faq` – frequently asked questions
- `history` – complete history of **ADDA** development
- `manual.doc` – source of this manual in MS Word format
- `manual.pdf` – this manual in PDF format
- `README` – brief description of **ADDA**

`input/` – default input files
- `tables/` – 10 auxiliary files with tables of integrals (§D.1)
- `alldir_params.dat` – parameters for integral scattering quantities (§B.3)
- `avg_params.dat` – parameters for orientation averaging (§B.2)
- `scat_params.dat` – parameters for grid of scattering angles (§B.4)

[8] `http://code.google.com/p/a-dda/source/browse/#svn/trunk`
[9] `svn checkout http://a-dda.googlecode.com/svn/trunk/`

`misc/` – additional files, not supported by the authors (§15).

`sample/` – sample output and other files

- `run000_sphere_g16m1_5/` – sample output directory (§C.3), contains `log` (§C.4), `mueller` (§C.5), and `CrossSec-Y` (§C.6)
- `test.pbs` – sample PBS script for MPI system (§5.2)
- `test.sge` – sample SGE script for MPI system (§5.2)
- `stdout` – standard output of a sample simulation (§C.2)

`src/`

- `Makefile`, `make_seq`, `make_mpi` – makefiles (§4)
- ADDAmain.c, CalculateE.c, calculator.c, cmplx.h, const.h, crosssec.c/h, comm.c/h, debug.c/h, fft.c, function.h, GenerateB.c, io.c/h, iterative.c, make_particle.c, matvec.c, memory.c/h, os.h, param.c/h, parbas.h, prec_time.c/h, Romberg.c/h, sinint.c, timing.c/h, types.h, vars.c/h – source and header files of **ADDA**
- `cfft99D.f` – source file for Temperton FFT (§13.2)
- `mt19937ar.c/h` – source and header files for Mersenne twister random generator (§8.5).

4 Compiling and Linking

4.1 Compiling ADDA

ADDA is written in C, but it contains one Fortran file (`cfft99D.f`) for built-in Fourier routines (they are only used if FFTW 3 is not installed) – see §13.2. On Unix systems **ADDA** can be easily compiled using provided `Makefile` – just type[10]

```
make seq
```

or

```
make mpi
```

while positioned in `src/` directory, for the sequential or MPI version respectively. Default compilers (`gcc` and `g77` for sequential, and `mpicc` and `mpif77` for MPI versions respectively) will be used together with maximum optimization flags. The makefiles are tested with `gcc` version 3.2.3 and higher. The compilation produces executable file named either `adda` or `adda_mpi` for the sequential or MPI version respectively.

It is possible to compile **ADDA** using Intel compiler.[11] Supplied makefiles contain all the necessary flags for optimal performance, one only need to change the value of flag `COMPILER` in `Makefile` to "`intel`". We have found from 10 to 20% improvement of speed for different **ADDA** parts when using `icc` 9.0 compared to `gcc` 3.4.4 (tests were performed on Intel Pentium 4 3.2 GHz and Intel Xeon 3.4 GHz and gave similar results). Therefore, we recommend using Intel compiler wherever possible. Note that it is freely available for non-commercial use on Linux systems.

The `Makefile` also contains all necessary optimization flags for Compaq[12] and IBM[13] compilers. However, we do not have constant access to systems with these compilers, so they are not as thoroughly tested as `gcc` and `icc`, described above. To use these compilers change the value of flag `COMPILER` in `Makefile` to "`compaq`" or "`ibm`" respectively.

[10] If you experience problems with `make`, use `gmake` instead.
[11] http://www.intel.com/cd/software/products/asmo-na/eng/compilers/
[12] http://h30097.www3.hp.com/dtk/compaqc_ov.html
[13] http://www-01.ibm.com/software/awdtools/xlcpp/

Other compilers may be added manually by modifying the makefiles according to the pattern used for predefined compilers. One should modify `Makefile` and, if necessary, `make_mpi` adding flags definitions after the line:

```
ifeq ($(COMPILER),other)
```

If you do so, please send the modified makefiles to the authors, so we may include them in newer versions of **ADDA**.

ADDA is fully ready for 64-bit mode, which allows it to use practically unlimited memory per single process. One only needs to provide 64-bit operating system and compiler environment. Compilation in non-native mode may be performed using additional compiler switches. For instance, if you wish to compile 32-bit binaries in 64-bit environment, add "`-m32`" to line starting "`CFLAGS +=`" below the compiler section in the `Makefile` (this works at least for both `gcc` and `icc`). Using 64-bit is recommended wherever possible.

It is important to note that starting from version 0.79 **ADDA** requires support of C99 standard.[14] More precisely, it uses a number of its features that are already supported by `gcc`, `icc`, and many other compilers. Some of this features may be easily implemented by a bunch of defines with any ANSI C compiler (e.g. functionality provided by `stdbool.h`), but others are hard to avoid without loosing robustness and portability of the code (e.g. `%z` argument to `printf`-family functions). Therefore, we decided to implement a hard check for compiler conformance to C99 during the compilation (the compilation will produce an error if this test fails). The significant drawback of it is that Microsoft and Borland compilers may have problems with compiling **ADDA**.[15] If you believe that your compiler supports all necessary features but for some reason do not declare itself conforming to C99 standard, you may override the test by uncommenting the line

```
CFLAGS += -DOVERRIDE_STDC_TEST
```

in the `Makefile`.

In order to compile **ADDA** with FFTW 3 support you need first to install the FFTW 3 package (§4.2), and to compile a parallel version (§13.3) MPI should be installed on your system (§4.3). **ADDA** is capable of running both on multiprocessor systems (clusters) and on multicore processors of a single PC, as well as on heterogeneous hardware, if MPI is installed. In this manual we will use a word "processor" both for individual single-core processors and for each core of a multi-core processor, if not explicitly stated otherwise.

There are several options that may be changed in the `Makefile` uncommenting the corresponding lines:

"`CFLAGS += -DDEBUG`" – debugging. Turns on additional information messages during the code execution.

"`CFLAGS += -DFFT_TEMPERTON`" – use FFT by C. Temperton (§13.2). Use it if you have problems installing the FFTW 3 package.

"`CFLAGS += -DPRECISE_TIMING`" – enable precise timing routines, which give extensive timing of all the computation parts of **ADDA**, useful for debugging or optimization studies (§14.2).

"`CFLAGS += -DNOT_USE_LOCK`" – do not use file locking for `ExpCount` at all; enable this flag if you get compilation errors due to unsupported operation system or experience permanent locks (§B.1).

"`CFLAGS += -DONLY_LOCKFILE`" – use lock file to lock `ExpCount`, but do not additionally lock this file to be robust even on network file systems. Enable this flag if you get warnings when running **ADDA** stating that your system do not supports advanced file locking (§B.1). This option is relevant only for POSIX systems.

[14] http://www.open-std.org/JTC1/SC22/WG14/www/standards

[15] We haven't tried it ourselves, so one can always give it a try. Please also let the authors know the results.

All compilation warnings are suppressed in public releases of **ADDA**. However, you may turn them on, by commenting a line

```
RELEASE = on
```

in `Makefile`. If you do so, please communicate obtained warnings to the authors.

Paths to FFTW 3 header and library files are determined by internal variables `FFTW3_INC_PATH` and `FFTW3_LIB_PATH` respectively. If FFTW 3 is installed globally on the system these variables are either not needed to be set (any value will be OK) or should be set to values of certain system variables, e.g. `FFTW_INC` and `FFTW_LIB`. If FFTW 3 is installed under user account, they should be set to `$HOME/include` and `$HOME/lib` respectively. If needed, uncomment corresponding lines in the `Makefile`. For MPI compilation you may either use compiler wrappers, e.g. `mpicc`, or the usual compiler, e.g. `gcc`, linking to MPI library manually. The choice is performed by uncommenting one of the lines starting with "`MPICC=`" in file `make_mpi`. If you link to MPI library manually, you may need to specify a path to it (and to the header file `mpi.h`) manually. For that uncomment and adjust lines starting with "`CFLAGS += -I`" and "`LFLAGS += -L`" accordingly. This lines are located in file `make_mpi`. after the line

```
ifeq ($(MPICC),$(CC))
```

Compilation on non-Unix systems is also possible, however it should be done manually – compile all the source files (with maximum possible optimizations) and link them into executable `adda`. The authors provide executable files, both sequential and parallel version, for 32-bit Windows, which are compiled with MinGW 5.1.4,[16] using the default `Makefile`. This executable is not included in the main package, but can be downloaded from

`http://code.google.com/p/a-dda/downloads`

It is distributed together with the dynamic link library for FFTW 3.1.2. The parallel executable was compiled linking to MPICH2 1.0.8p1,[17] and requires it to be installed on the system (§4.3). However, it may also work with other MPI implementations (try it at your own risk). The Win32 package is the fastest way to start with **ADDA**, however it has limited functionality and cannot be optimized for a particular hardware. It is important to note that this package contains only Windows-specific files; therefore, it should be used together with the main package. We plan to provide binaries for 64-bit Windows as well in the near future.

So far as we know there are only two operating-system-dependent aspects of **ADDA**: precise timing (§14.2), and file locking (§B.1). Both are optional and can be turned off by compilation flags. However these features should be functional for any Windows or POSIX-compliant (Unix) operating system.

4.2 Installing FFTW 3

The installation of FFTW 3 package[18] on any Unix system is straightforward and therefore is *highly recommended*, since it greatly improves the performance of **ADDA**. The easiest is to install FFTW 3 for the entire system, using `root` account.[19] However, it also can be installed under any user account as follows:

- Download the latest version of FFTW 3 from
  `http://www.fftw.org/download.html`
- Unpack it, cd into its directory, and type
  ```
  ./configure --prefix=$HOME [--enable-sse2|--enable-k7]
  ```

[16] `http://www.mingw.org/`, it is based on gcc 3.4.5.
[17] `http://www.mcs.anl.gov/mpi/mpich/`
[18] `http://www.fftw.org`
[19] Details are in `http://www.fftw.org/fftw3_doc/Installation-on-Unix.html`

where "`--enable`" options are specialized for modern Intel or AMD processors respectively.[19] Then type

```
make
make install
```

- Modify the initialization of internal variables `FFTW3_INC_PATH` and `FFTW3_LIB_PATH` in the `Makefile`, as described in §4.1.

Installation of FFTW 3 on non-Unix systems is slightly more complicated. It is described in `http://www.fftw.org/install/windows.html`. It is important to note that FFTW 3 installation is not required to use Windows executables included in the **ADDA** package.

4.3 Installing MPI

If you plan to run **ADDA** on a cluster, MPI is probably already installed on your system. You should consult someone familiar with the particular MPI package. **ADDA**'s usage of MPI is based on the MPI 1.2 standard,[20] and it should work with any implementation that is compliant with this or higher versions of the standard. Version of MPI standard is checked for conformity by **ADDA** both during compilation and at runtime.

If you plan to run a parallel version of **ADDA** on a single computer, e.g. using multicore processor, you need first to install some implementation of MPI. Installation instruction can be found in the manual of a particular MPI package. In the following we describe a single example of installation of MPICH 2 package on Windows.

- Download the installer of MPICH 2 for your combination of Windows and hardware from `http://www.mcs.anl.gov/mpi/mpich/`, and install it.
- Specify paths to `include` and `lib` subdirectories of MPICH 2 directory in file `make_mpi`, as described in §4.1. Add `bin` subdirectory to Windows environmental variable `PATH`.
- MPICH needs user credentials to run on Windows. To remove asking for them at every MPI run, run a GUI utility `wmpiregister.exe` or type:

  ```
  mpiexec –register
  ```

  You will be asked for your credentials, which then will be stored encrypted in the Windows registry.
- First time you run `mpiexec` you probably will be prompted by the Windows firewall, whether to allow `mpiexec` and `smpd` to access network. If you plan to run **ADDA** only on a single PC, you may block them. Otherwise, you should unblock them.

MPICH 2 allows much more advanced features. For instance, one may combine several single- and multi-core Windows machines in a cluster. However, we do not discuss it here.

ADDA will run on any hardware compatible with MPI, but, in principle, it may run more efficiently on hardware with shared memory address space, e.g. multi-core PCs. However, such hardware also has its drawbacks, e.g. the performance may be limited by memory access speed. A few tests performed on different computers showed that currently using two cores of a dual-core PC results in computational time from 60% to 75% of that for sequential execution on the same machine. We plan to optimize **ADDA** specifically for such hardware.

[20] `http://www.mpi-forum.org`

5 Running ADDA

5.1 Sequential mode

The simplest way to run **ADDA** is to type

```
adda
```
[21]

while positioned in a directory, where the executable is located. **ADDA** will perform a sample simulation (sphere with size parameter 3.367, refractive index 1.5, discretized into 16 dipoles in each direction) and produce basic output (§12, §C). The output directory and terminal output (`stdout`) should look like examples that are included in the distribution: `sample/run000_sphere_g16m1_5` and `sample/stdout` respectively. **ADDA** takes most information specifying what and how to calculate from the command line, so the general way to call **ADDA** is

```
adda -<par1> <args1> -<par2> <args2> …
```

where `<par>` is an option name (starting with a letter), and `<args>` is none, one, or several arguments (depending on the option), separated by spaces. `<args>` can be both text or numerical. How to control **ADDA** by proper command line options is thoroughly described in the following sections; the full reference list is given in §A. If you prefer a quick hands-on start with **ADDA** before looking at all possible options, you are welcome to a tutorial described in §G. Quick help is available by typing

```
adda -h
```

For some options input files are required, they are described in §B. It is recommended to copy the contents of the directory `input/` of the distribution (which contains examples of all input files) to the directory where **ADDA** is executed. All the output produced by **ADDA** is described in §C. Version of **ADDA**, compiler used to build it,[22] width of memory access registers (32 or 64 bits), and copyright information are available by typing

```
adda –V
```

5.2 Parallel mode

On different systems MPI is used differently, you should consult someone familiar with MPI usage on your system. However, running on a multi-core PC is simple, just type

```
mpiexec –n <N> ./adda_mpi …
```

where all **ADDA** command line options are specified in the end of the line, and `<N>` stands for the number of cores. Actually, `<N>` is the number of threads created by MPI implementation, and it should not necessarily be equal to the number of cores. However, this choice is recommended. MPICH 2 allows one to force local execution of all threads by additionally specifying command line option "`-localonly`" after `<N>`. However, this may be different for other MPI implementations.

Running on a cluster is usually not that trivial. For example, our main test platform is the Dutch national compute cluster LISA.[23] There, as on many other parallel computers, PBS (portable batch system)[24] is used to schedule jobs. To schedule a job one should first write a shell script and then submit it. An example of such PBS script is included in the distribution (`sample/test.pbs`). It is important to note that usually output of the batch job (both `stdout` and `stderr`) is saved to file. The name of the file usually contains the job id number given by the system. The same number appears in the directory name (§C.3).

---

[21] If current directory is not in the `PATH` system variable you should type "`./adda`". It may also differ on non-Unix systems, e.g. under Windows you should type "`adda.exe`". Moreover, the name of executable is different for MPI version (§5.2). This applies to all examples of command lines in this manual.

[22] Only a limited set of compilers is recognized (currently: Borland, Compaq, GNU, Intel, Microsoft).

[23] https://subtrac.sara.nl/userdoc/wiki/lisa/description

[24] http://www.openpbs.org/

Another batch system is SGE (Sun grid engine).[25] We do not give a description of it here, but provide a sample script to run **ADDA** using SGE (`sample/test.sge`). One can easily modify it for a particular task.

6 Applicability of DDA

The principal advantage of the DDA is that it is completely flexible regarding the geometry of the scatterer, being limited only by the need to use dipole size d small compared to any structural length in the scatterer and the wavelength λ. A number of studies devoted to the accuracy of DDA results exist, e.g. [4-6,10,11,17-22], most of them are reviewed in [12]. Here we only give a brief overview.

The rule of thumb for particles with size comparable to the wavelength is: "10 dipoles per wavelength inside the scatterer", i.e. size of one dipole is

$$d = \lambda/10|m|, \tag{1}$$

where m is refractive index of the scatterer. That is the default for **ADDA** (§8.2). The expected accuracy of cross sections is then several percents (for moderate m, see below). With increasing m the number of dipoles that is used to discretize the particle increases, moreover the convergence of the iterative solver (§13.1) becomes slower. Additionally, accuracy of the simulation with default dipole size becomes worse, and smaller dipoles (hence larger number of them) must be used to improve it. Therefore, it is accepted that the refractive index should satisfy

$$|m-1| < 2. \tag{2}$$

Higher m can be simulated accurately; however, required computer resources rapidly increase with m. Nevertheless, modern DDA formulations (§11) alleviate this problem and make higher refractive indices accessible. But application of DDA in this regime is studied much less thoroughly than for moderate refractive indices.

When larger scatterers are considered (volume-equivalent size parameter $x > 10$) the rule of thumb still applies. However, it does not describe well the dependence on m. When employing the rule of thumb, errors do not significantly depend on x, but do significantly increase with m [17]. However, the simulation data for large scatterers is limited; therefore, it is hard to propose any simple method to set dipole size instead of the rule of thumb. The maximum reachable x and m are mostly determined by the available computer resources (§7).

DDA also is applicable to particles smaller than the wavelength, e.g. nanoparticles. In some respects, it is even simpler than for larger particles, since many convergence problem for large m are not present for small scatterers. However, there is additional requirement for d – it should adequately describe the shape of the particle [this requirement is relevant for any scatterers, but for larger scatterers it is usually automatically satisfied by Eq. (1)]. For instance, for a sphere (or similar compact shape) it is recommended to use at least 10 dipoles along the smallest dimension, no matter how small is the particle. Smaller dipoles are required for irregularly shaped particles and/or large refractive index.

To conclude, it is hard to estimate *a priori* the accuracy of DDA simulation for a particular particle shape, size, and refractive index, although the papers cited above do give a hint. If you run a single DDA simulation, there is no better alternative than to use rule of thumb and hope that the accuracy will be similar to that of the spheres, which can be found in one of the benchmark papers (e.g. [17,21,10]). However, if you plan to run a series of simulations for similar particles, especially outside of the usual DDA application domain [Eq. (2)], it is highly recommended to perform a small accuracy study. For that choose a single test particle and perform DDA simulations with different d, both smaller and larger

[25] http://gridengine.sunsource.net/

than proposed by the rule of thumb. Looking at how the results change with decreasing d, you can estimate d required for your particular accuracy. You may also make the estimation much more rigorous by using extrapolation technique, as described by Yurkin *et al.* [23].

Finally, it is important to note that DDA in its original form is derived for finite particles in vacuum. However, it is perfectly applicable to finite particles embedded in a homogeneous non-absorbing dielectric medium (refractive index m_0). To account for the medium one should replace the particle refractive index m by the relative refractive index m/m_0, and the wavelength in the vacuum λ by the wavelength in the medium λ/m_0. All the scattering quantities produced by DDA simulation with modified m and λ are then the correct ones for the initial scattering problem. This applies to any DDA code and **ADDA** in particular.

DDA can not be directly applied to particles, which are not isolated from other scatterers. In particular, it can *not* be applied to particles located near the infinite dielectric plane surface. Although a modification of DDA is possible to solve this problem rigorously [24,25], it requires principal changes in the DDA formulation and hence in the internal structure of the computer code. Therefore, this feature will not be implemented in **ADDA** in the near future. However, one may approach this problem using the current version of **ADDA**. 1) One may completely ignore the surface, this may be accurate enough if particle is far from the surface and the refractive index change when crossing the surface is small. 2) A little more accurate approach is to consider the "incident field" in **ADDA** as a sum of incident and reflected plane waves. However, currently such incident field is not implemented, therefore it need first to be coded into **ADDA**. 3) Another approach is to take a large computational box around the particle, and discretize the surface that falls into it. This is rigorous in the limit of infinite size of computational box, but requires much larger computer resources than that for the initial problem. It also requires one to specify the shape files for all different sizes of the computational box.

It is important to note that for some problem parameters other methods can be clearly superior compared to DDA. For instance, see §E.3 for comparison of DDA performance to that of the finite difference time domain method.

7 System Requirements

Computational requirements of DDA primarily depend on the size of computational grid, which in turn depends on the size parameter x and refractive index m of the scatterer (§8.2). The memory requirements of **ADDA** depend both on the total number of dipoles in a computational box (N) and number of real (non-void) dipoles (N_{real}); it also depends on number of dipoles along x-axis (n_x) and number of processors used (n_{p}). Total memory requirements (in bytes) are approximately

$$mem = \left[288 + 384\, n_{\text{p}}/n_x \ \left(+192/n_{\text{p}}\right)\right] N + \left[271 \ \ (+144)\right] N_{\text{real}}, \tag{3}$$

where additional memory (in round brackets) proportional to N is required only in parallel mode, and proportional to N_{real} – only for the QMR and Bi-CGStab iterative solvers (§13.1). Moreover, in multiprocessor mode part proportional to N may be slightly higher (see §13.2 for details). The memory requirements of each processor depends on the partition of the computational grid over the processors that is generally not uniform (see §0). Total memory used by **ADDA** and maximum per one processor are shown in `log` (see §C.4). It is important to note that *double* precision is used everywhere in **ADDA**. This requires more memory (compared to single precision), but it helps when convergence of the iterative solver is very slow and machine precision becomes relevant (that is the case for large simulations) or when very accurate results are desired, as in [23].

ADDA may optimize itself during runtime for either maximum speed or minimum memory usage. It is controlled by a command line option

`-opt {speed|mem}`

By default, speed optimization is used. Currently the difference in performance is very small, but it will increase during future **ADDA** development. A command line option

`-prognosis`

can be used to estimate the memory requirements without actually performing the allocation of memory and simulation.[26] It also implies `-test` option (§C.3).

Simulation time (see §13 for details) consists of two major parts: solution of the linear system of equations and calculation of the scattered fields. The first one depends on the number of iterations to reach convergence, which mainly depends on the size parameter, shape and refractive index of the scatterer, and time of one iteration, which depends only on N scaling as O(Nln$N$) (see §13.2). Time for calculation of scattered fields is proportional to N_{real}, and is usually relatively small if scattering is only calculated in one plane. However, it may be significant when a large grid of scattering angles is used (§12.1, §12.2). Employing multiple processors brings the simulation time down almost proportional to the number of processors (see §13.3). To facilitate very long simulations checkpoints can be used to break a single simulation into smaller parts (§13.4).

For example, on a modern desktop computer[27] (P4-3.2 GHz, 2 Gb RAM) it is possible to simulate light scattering by particles[28] up to $x = 35$ and 20 for $m = 1.313$ and 2.0 respectively (simulation times are 20 and 148 hours respectively[29]). The capabilities of **ADDA** v.0.76 for simulation of light scattering by spheres using 32 nodes of LISA[23] (each dual P4-3.4 GHz with 4Gb RAM) were reported in [17]. Here we present only Figs. 1 and 2, showing the maximum reachable x versus m and simulation time versus x and m respectively. For instance, light scattering by a homogenous sphere with $x = 160$ and $m = 1.05$ can be simulated in only 1.5 hours.

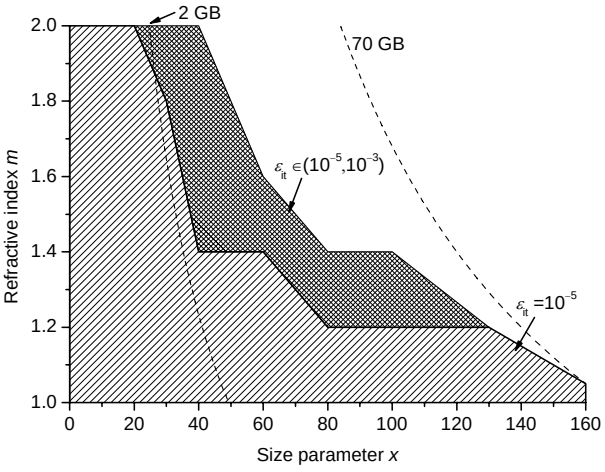

Fig. 1. Capabilities of **ADDA** v.0.76 for spheres with different x and m. The striped region corresponds to full convergence and densely hatched region to incomplete convergence. The dashed lines show two levels of memory requirements for the simulation. Adapted from [17].

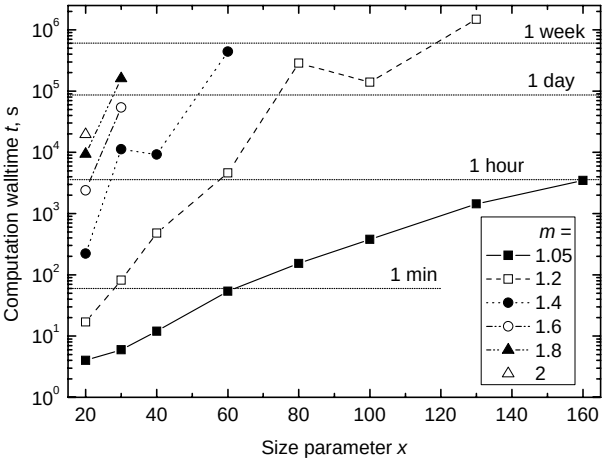

Fig. 2. Total simulation wall clock time (on 64 processors) for spheres with different x and m. Time is shown in logarithmic scale. Horizontal dotted lines corresponding to a minute, an hour, a day, and a week are shown for convenience. Adapted from [17].

[26] Currently this option does need a certain amount of RAM, about $11(N+N_{\text{real}})$ bytes. It enables saving of the particle geometry in combination with `-prognosis`.

[27] At time of writing, spring 2006.

[28] Shown values are for spheres, for other shapes they may vary. Only one incident polarization was calculated, execution time for non-symmetric shapes (§8.7) will be doubled.

[29] Compiled with Intel C++ compiler version 9.0.

8 Defining a Scatterer

8.1 Reference frames

Three different reference frames are used by **ADDA**: laboratory, particle, and incident wave reference frames. The laboratory reference frame is the default one, all input parameters and other reference frames are specified relative to it. **ADDA** simulates light scattering in the particle reference frame, which naturally corresponds to particle geometry and symmetries, to minimize the size of computational grid (§8.2), especially for elongated or oblate particles. In this reference frames computational grid is build along the coordinate axes. The incident wave reference frame is defined by setting the z-axis along the propagation direction. All scattering directions are specified in this reference frame.

The origins of all reference frames coincide with the center of the computational box (§8.2). By default, both particle and incident wave reference frames coincide with the laboratory one, however they can be made different by rotating the particle (§9) or specifying different propagation direction of the incident beam (§10.1) respectively.

8.2 The computational grid

ADDA embeds any scatterer in a rectangular computational box, which is divided into identical cubes.[30] Each cube is called a "dipole", its size should be much smaller than a wavelength. The flexibility of the DDA method lies in its ability to naturally simulate the scattering of any arbitrarily shaped and/or inhomogeneous scatterer, because the optical properties (refractive index, §8.3) of each dipole can be set independently. There are a few parameters describing the simulation grid: size of one dipole (cube) d, number of dipoles along each axis n_x, n_y, n_z, total size (in μm) of the grid along each axis D_x, D_y, D_z, volume-equivalent radius r_{eq}, and incident wavelength λ. However not all of them are independent. **ADDA** allows one to specify all three grid dimensions n_x, n_y, n_z as corresponding arguments to the command line option

```
-grid <nx> [<ny> <nz>]
```

however in most cases `<ny>` and `<nz>` can be omitted. Then n_y, n_z are automatically determined by n_x based on the proportions of the scatterer (§8.4). If particle geometry is read from a file (§8.3) all the grid dimensions are initialized automatically.[31] Because of the internal structure of the **ADDA** all the dimensions are limited to be even. If odd grid dimension is specified by any input method, it is automatically incremented. In this case **ADDA** produces a warning to avoid possible ambiguity. If the `-jagged` option is used the grid dimension is effectively multiplied by the specified number (§8.3).

ADDA allows also specifying size parameter of the entire grid kD_x (k is free space wave vector) or $x = kr_{ef}$, using three command line options:

```
-lambda <arg>
-size <arg>
-eq_rad <arg>
```

which specify (in μm) λ, D_x and r_{ef} respectively. By default $\lambda = 2\pi$ μm, then `-size` determines kD_x and `-eq_rad` sets x. The last two are interconnected by

$$x = \left(\frac{3}{4\pi} f_{vol}\right)^{1/3} kD_x, \tag{4}$$

[30] The equally spaced cubical grid is required for the FFT-based method (§13.2) that is used to accelerate matrix-vector products in iterative solution of the DDA linear system (§13.1). Otherwise DDA computational requirements are practically unbearable.

[31] Specifying all three dimensions (or even one when particle geometry is read from file) make sense only to fix these dimensions (larger than optimal) e.g. for performance studies.

where f_{vol} is ratio of particle to computational grid volumes, which is known analytically for many shapes (§8.4). It is important to note that D_x denotes the size of *possibly adjusted* computational grid. Although **ADDA** will warn an user of possible ambiguities, it is not recommended to use `-size` command line option for shapes read from file, when inherent n_x of this shape is not guaranteed to be even. It may cause a modeled scatterer to be slightly smaller than originally intended. Some shapes define the absolute particle size themselves (§8.4). However, the size given in the command line (by either `-size` or `-eq_rad`) overrides the internal specification and the shape is scaled accordingly.

The size parameter of the dipole is specified by the parameter "dipoles per lambda" (dpl)

$$\mathrm{dpl} = \frac{\lambda}{d} = \frac{2\pi}{kd}, \tag{5}$$

which is given to the command line option

```
-dpl <arg>
```

dpl does not need to be an integer, any real number can be specified.

ADDA will accept at most two parameters from: dpl, n_x, kD_x, and x since they depend on each other by Eq. (4) and

$$kD_x \cdot \mathrm{dpl} = 2\pi \cdot n_x . \tag{6}$$

Moreover, specifying a pair of kD_x and x is also not possible. If any other pair from these four parameters is given on the command line (n_x is also defined if particle geometry is read from file) the other two are automatically determined from the Eqs. (4) and (6). If the latter is n_x, dpl is slightly increased (if needed) so that n_x exactly equals an even integer. There is one exception: a pair of x and dpl can only be used for shapes, for which f_{vol} can be determined analytically (§8.4), because numerical evaluation of f_{vol} is only possible when particle is already discretized with a certain n_x. If less than two parameters are defined dpl or/and grid dimension are set by default.[32] The default for dpl is $10|m|$ [cf. Eq. (1)], where m is the maximum (by absolute value) refractive index specified by the `-m` option (or the default one, §8.3). The default for n_x is 16 (possibly multiplied by `-jagged` value). Hence, if only `-size` or `-eq_rad` is specified, **ADDA** will automatically discretize the particle, using the default dpl (with the exception discussed above for x). This procedure may lead to very small n_x, e.g. for nanoparticles, hence **ADDA** ensures that n_x is at least 16, when it is auto-set from default dpl. However, if dpl is specified in the command line, **ADDA** puts absolute trust in it and leaves all the consequences to the user.

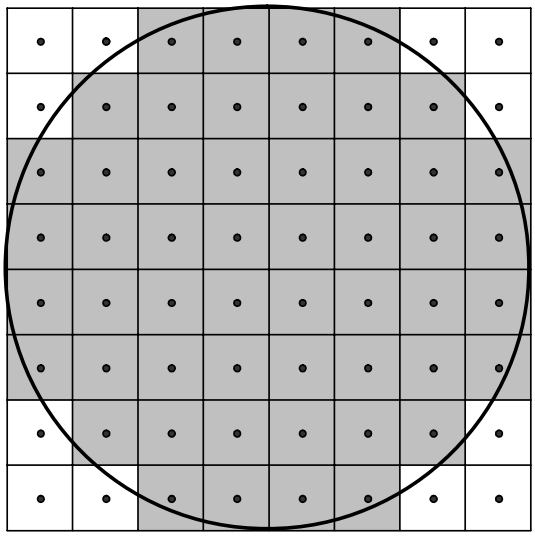

Fig. 3. Example of dipole assignment for a sphere (2D projection). Assigned dipoles are gray and void dipoles are white.

8.3 Construction of a dipole set

After defining the computational grid (§8.2) each dipole of the grid should be assigned a refractive index (a void dipole is equivalent to a dipole with refractive index equal to 1). This can be done automatically for a number of predefined shapes or in a very flexible way – specifying a scatterer geometry in a separate input file. Predefined shapes are described in detail in §8.4. The dipole is

[32] If dpl is not defined, it is set to the default value. Then, if still less than two parameters are initialized, grid dimension is also set to the default value.

assigned to the scatterer if its center belongs to it (see Fig. 3 for an example). When the scatterer consists of several domains, e.g. coated sphere, the same rule applies to each domain. **ADDA** has an option to slightly correct the dipole size (or equivalently dpl) to ensure that the volume of the dipole representation of the particle is exactly correct (Fig. 4), i.e. exactly corresponds to x. This is believed to increase the accuracy of DDA, especially for small scatterers [4]. By default, **ADDA** will try to ensure correct volume either using specified x or calculating it from specified kD_x (if shape permits). However, it introduces a small inconvenience that the size of the computational grid is not exactly equal to the size of the particle. Such a volume correction can be turned off by command line option

```
-no_vol_cor
```

In this case **ADDA** tries to match the size of the particle using specified kD_x or calculating it from specified x (if shape permits). Moreover, **ADDA** then determines the "real" value of x numerically from the volume of the dipole representation. Its value is shown in `log` file (§C.4) and is further used in **ADDA**, e.g. for normalization of cross sections (§12.2), although it may slightly differ from specified or analytically derived x.

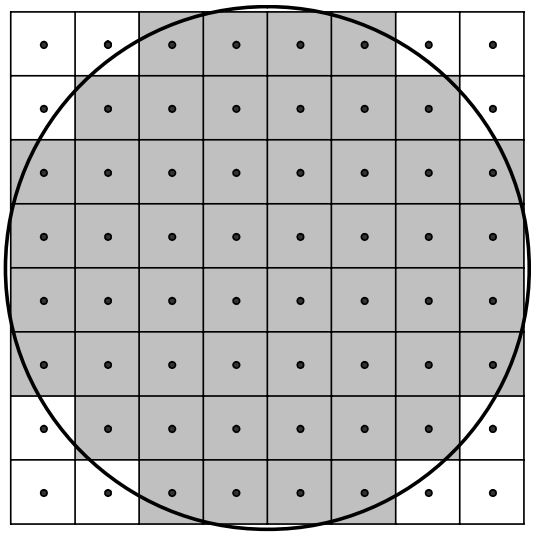

Fig. 4. Same as Fig. 3 but after the "dpl correction".

To read a particle geometry from a file, specify the file name as an argument to the command line option

```
-shape read <filename>
```

This file specifies all the dipoles in the simulation grid that belongs to the particle (possibly several domains with different refractive indices). Both **ADDA** text formats and DDSCAT 6.1 format, corresponding to its shape option `FRMFIL` and output of `calltarget` utility [26], are supported,[33] as described in §B.5. Dimensions of the computational grid are then initialized automatically.

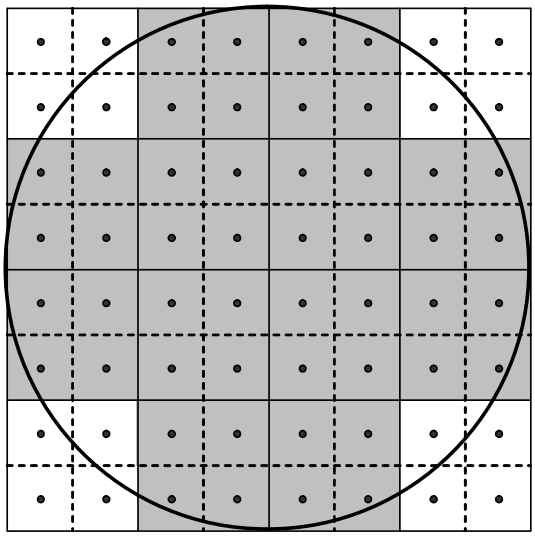

Fig. 5. Same as Fig. 3 but with "`-jagged`" option enabled ($J = 2$). The total grid dimension is the same.

Sometimes it is useful to describe particle geometry in a coarse way by bigger dipoles (cubes), but then use smaller dipoles for the simulation itself.[34] **ADDA** enables it by the command line option

```
-jagged <arg>
```

which specifies a multiplier J. For construction of the dipole set big cubes ($J \times J \times J$ dipoles) are used (Fig. 5) – center of big cubes are tested for belonging to a particle's domain. All grid dimensions are multiplied by J. When particle geometry is read from file it is considered to be a configuration of big cubes, each of them is further subdivided into J^3 dipoles.

[33] Support of DDSCAT format has certain limitations for anisotropic scatterers.

[34] This option may be used e.g. to directly study the shape errors in DDA (i.e. caused by imperfect description of the particle shape) [23].

ADDA includes a granule generator, which can automatically fill specified domain with granules of a certain size. It is described in details in §8.5.

The last parameter to completely specify a scatterer is its refractive index. Refractive indices are given on the command line

```
-m {<m1Re> <m1Im> […]|<m1xxRe> <m1xxIm> <m1yyRe> <m1yyIm>
<m1zzRe> <m1zzIm> […]}
```

Each pair of arguments specifies the real and imaginary part[35] of the refractive index of the corresponding domain (first pair corresponds to domain number 1, etc.). Command line option

```
-anisotr
```

can be used to specify that refractive index is anisotropic, then three refractive indices correspond to one domain. They are the diagonal elements of the refractive index tensor in the particle reference frame (§8.1). Currently **ADDA** supports only diagonal refractive index tensors; moreover, the refractive index must change discretely. Anisotropy can not be used with CLDR polarizability (§11.1) and all SO formulations (§11.1, §11.2, §11.3), since they are derived assuming isotropic refractive index, and can not be easily generalized. Use of anisotropic refractive index cancels the rotation symmetry if its x and y-components differ. Limited testing of this option was performed for Rayleigh anisotropic spheres.

The maximum number of different refractive indices (particle domains) is defined at compilation time by the parameter `MAX_NMAT` in the file `const.h`. By default it is set to 15. The number of the domain in the geometry file (§B.5) exactly corresponds to the number of the refractive index. Numbering of domains for the predefined shapes is described in §8.4. If no refractive index is specified, it is set to 1.5, but this default option works only for one-domain isotropic scatterers. Refractive indices may be not specified when `-prognosis` option is used. Currently **ADDA** produces an error if any of the given refractive index equals to 1. It is planned to improve this behavior to accept such refractive index and automatically make corresponding domain void. This can be used, for instance, to generate spherical shell shape using standard option `–shape coated`. For now, one may set refractive index to the value very close to 1 for this purpose, e.g. equal to 1.00001.

ADDA is able to save the constructed dipole set to a file if the command line option

```
-save_geom [<filename>]
```

is specified. `<filename>` is an optional argument (it is a path relative to the output directory, §C.3). If it is not specified, **ADDA** names the output file `<type>.geom`. `<type>` is shape name – a first argument to the `-shape` command line option, see above and §8.4, possibly with addition of `_gran` (§8.5). The format is determined by the command line option

```
-sg_format {text|text_ext|ddscat}
```

First two are **ADDA** default formats for single- and multi-domain particles respectively. DDSCAT 6.1 format corresponds to its shape option `FRMFIL` and output of `calltarget` utility [26]. `text` is automatically changed to `text_ext` for multi-domain particles. Output formats are fully compatible with the input ones[36] (see §C.10 for details). The values of refractive indices are not saved (only domain numbers). This option can be combined with `-prognosis`, then no DDA simulation is performed but the geometry file is generated.

8.4 Predefined shapes

Predefined shapes are initialized by the command line option

```
-shape <name> <args>
```

[35] **ADDA** uses exp(-iωt) convention for time dependence of harmonic electric field, therefore absorbing materials have *positive* imaginary part.

[36] with similar limitations of DDSCAT format for anisotropic scatterers.

Table 1. Brief description of arguments, symmetries (§8.7), and availability of analytical value of f_{vol} for predefined shapes. Shapes and their arguments are described in the text. "±" means that it depends on the arguments.

<name>	<args>	dom.[a]	symY[b]	symR[c]	f_{vol}	size[d]
`axisymmetric`	filename	1	+	+	–	+
`box`	[y/x, z/x]	1	+	±	+	–
`capsule`	h/d	1	+	+	+	–
`coated`	d_{in}/d, [x/d, y/d, z/d]	2	±	±	+	–
`cylinder`	h/d	1	+	+	+	–
`egg`	ε, ν	1	+	+	+	–
`ellipsoid`	y/x, z/x	1	+	±	+	–
`line`	–	1	–	–	–	–
`rbc`	h/d, b/d, c/d	1	+	+	–	–
`sphere`	–	1	+	+	+	–
`spherebox`	d_{sph}/D_x	2	+	+	+	–

[a] number of domains.
[b] symmetry with respect to reflection over xz-plane.
[c] symmetry with respect to rotation by 90° over z-axis.
[d] whether a shape defines absolute size of the particle.

where `<name>` is a name of the predefined shape. The size of the scatterer is determined by the size of the computational grid (D_x, §8.2); `<args>` specify different dimensionless aspect ratios or other proportions of the particle shape. However, some shapes define the absolute size themselves, which can be overridden (§8.2).

In the following we describe all the supported predefined shapes, all the reference information is summarized in Table 1. "`axisymmetric`" is an axisymmetric homogeneous shape, defined by its contour in ρz-plane of the cylindrical coordinate system. Its symmetry axis coincides with the z-axis, and the contour is read from file (format described in §B.6). "`box`" is a homogenous cube (if no arguments are given) or a rectangular parallelepiped with edge sizes x, y, z. "`capsule`" is a cylinder with height (length) h and diameter d with half-spherical caps on both ends (its axis of symmetry coincides with the z-axis). Total height of the capsule is $h + d$. "`coated`" is a sphere with a spherical inclusion; outer sphere has a diameter d (first domain[37]). The included sphere has a diameter d_{in} (optional position of the center: x, y, z). "`cylinder`" is a homogenous cylinder with height (length) h and diameter d (its axis of symmetry coincides with the z-axis). "`egg`" is an axisymmetric (over z-axis) biconcave homogenous particle, which surface is given as

$$r^2 + \nu r z - (1-\varepsilon) z^2 = a^2, \tag{7}$$

where r is the radius in spherical coordinates and a is scaling factor, which can be derived from egg diameter (maximum width perpendicular to z-axis) or volume.[38] "`ellipsoid`" is a homogenous general ellipsoid with semi-axes x, y, z. "`line`" is a line along the x-axis with the width of one dipole. "`rbc`" is a Red Blood Cell, an axisymmetric (over z-axis) biconcave homogenous particle, which is characterized by diameter d, maximum and minimum width h, b, and diameter at the position of the maximum width c. Its surface is defined as

$$\rho^4 + 2S\rho^2 z^2 + z^4 + P\rho^2 + Qz^2 + R = 0, \tag{8}$$

[37] The order of domains is important to assign refractive indices specified in the command line (§8.3).
[38] Egg shape was proposed in [52]. Actual expressions connecting diameter, height, a, and volume are cumbersome; they can be found in comments in the code. Center of the reference frame in Eq.(7) is shifted along z-axis relative to the center of the computational grid to ensure that North and South Poles of the egg are symmetric over the latter center. This is done to minimize the size of the computational grid.

where ρ is the radius in cylindrical coordinates ($\rho^2 = x^2 + y^2$). P, Q, R, and S are determined from the values of the parameters given in the command line.[39] "`sphere`" is a homogenous sphere (used by default). "`spherebox`" – a sphere (diameter d_{sph}) in a cube (size D_x, first domain). For multi-domain shapes f_{vol} is based on the total volume of the particle.

We are currently working to increase the number of predefined shapes. Moreover, adding of a new shape is straightforward for anyone who is familiar with C programming language. The procedure is described in §F.1.

8.5 Granule generator

Granule generator is enabled by the command line option

```
-granul <vol_frac> <diam> [<dom_number>]
```

which specifies that one particle domain should be randomly filled with spherical granules with specified diameter `<diam>` and volume fraction `<vol_frac>`. Domain number to fill is given by the last optional argument (default is the first domain). Total number of domains is then increased by one; the last is assigned to the granules. Suffix "`_gran`" is added to the shape name and all particle symmetries (§8.7) are cancelled.

A simplest algorithm is used: to place randomly a sphere and see whether it fits in the given domain together with all previously placed granules. The only information that is used about some of the previously place granules is dipoles occupied by them, therefore intersection of two granules is checked through the dipoles, which is not exact, especially for small granules. However it should not introduce errors larger than those caused by the discretization of granules. Moreover, it allows considering arbitrary complex domains, which is described only by a set of occupied dipoles. This algorithm is unsuitable for high volume fractions, it becomes very slow and for some volume fractions may fail at all (depending on the size of the granules critical volume fractions is 30-50%). Moreover, statistical properties of the obtained granules distribution may be not perfect; however, it seems good enough for most applications. To generate random numbers we use Mersenne twister,[40] which combines high speed with good statistical properties [27]. The granule generator was used to simulate light scattering by granulated spheres by Yurkin *et al.* [28].

If volume correction (§8.3) is used diameter of the granules is slightly adjusted to give exact *a priori* volume fraction. *A posteriori* volume fraction is determined based on the total number of dipoles occupied by granules and is saved to `log` (§C.4). It is not recommended to use granule diameter smaller than the size of the dipole, since then dipole grid can not adequately represent the granules, even statistically. **ADDA** will show a warning in that case; however, it will perform simulation for any granule size.

Currently the granule generator does not take into account `-jagged` option. We plan to rewrite the implementation of the latter; that will fix this problem. For now one may save a geometry file for a particle model scaled to $J = 1$ and then load it using any desired J. The same trick can be used to fill different particle domains and/or using different sizes of granules. To do it the complete operation should be decomposed into elementary granule fills, which should be interweaved with saving and loading of geometry files.

Coordinates of the granules (its centers) can be saved to a file `granules` (§C.11) specifying command line option

```
-store_grans
```

This provides more accurate information about the granulated particle than its dipole representation, which can be used e.g. to refine discretization.

[39] RBC shape is based on [53], and it is similar to the RBC shape used in [54].
[40] http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html

8.6 Partition over processors in parallel mode

To understand the parallel performance of **ADDA** it is important to realize how a scattering problem is partitioned over the processors. It is done in a simple and robust, but not necessarily the most efficient way. Both the computational grid and the scatterer are partitioned in slices parallel to *xy*-plane (in another words, partition is performed over the *z*-axis); each processor contains several of these slices. For the FFT-based task (§13.2) – the matrix-vector product that takes most of the time of iterative solution (§13.1) – the whole grid is partitioned[41]. The partition over the *z*-axis is optimal for this task if n_z divides the number of processors (at least approximately).

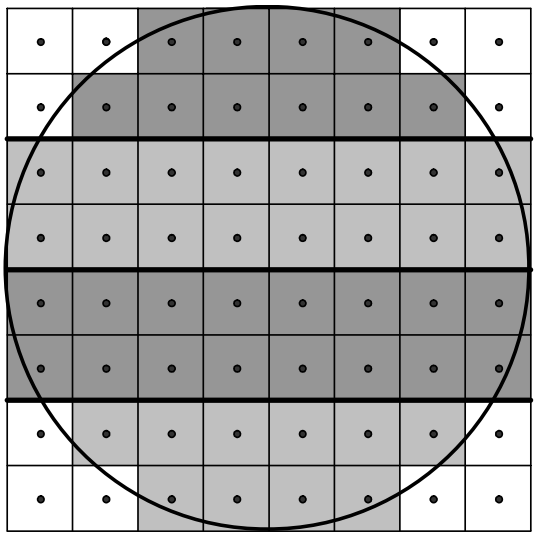

Fig. 6. Same as Fig. 3 but partitioned over 4 processors (shown in different shades of gray).

The partition of the scatterer itself also benefits from the latter condition, however it is still not optimal for most of the geometries,[42] i.e. the number of non-void dipoles is different for different processors (Fig. 6). This partition is relevant for the computation of the scattered fields, hence its non-optimality should not be an issue in most cases. However, if large grid of scattering angles is used (§12.1, §12.2), the parallel performance of the **ADDA** may be relatively low (the total simulation time will be determined by the maximum number of real dipoles per processor).[43]

The conclusion of this section is that careful choice of n_z and number of the processors (so that the former divides the latter) may significantly improve the parallel performance. **ADDA** will work fine with any input parameters, so this optimization is left to the user. Consider also some limitations imposed on the grid dimensions by the implemented FFT routines (§13.2).

8.7 Particle symmetries

Symmetries of a light scattering problem are used in **ADDA** to reduce simulation time. All the symmetries are defined for the default incident beam (§10). If the particle is symmetric with respect to reflection over the *xz*-plane, only half of the scattering *yz*-plane is calculated (scattering angle from 0° to 180°, §12.1). If the particle is symmetric with respect to rotation by 90° over the *z*-axis, the Mueller matrix in the *yz*-plane (§12.1) can be calculated from the calculation of the internal fields for just one incident polarization (*y* polarization is used). The second polarization is then equivalent to the first one but with scattering in *xz*-plane (in negative direction of *x*-axis). The symmetries are automatically determined for all the predefined shapes (§8.4). Some or all of them are automatically cancelled if not default beam type and/or direction (§10), anisotropic refractive index (§8.3), or granule generator (§8.5) are used.

Use of symmetry can be controlled by the command line option:

```
-sym {auto|no|enf}
```

First option corresponds to the default behavior described before, while `no` and `enf` specify to never use or enforce symmetry respectively. Use the latter with caution, as it may lead to

[41] More exactly, the grid is doubled in each dimension and then partitioned (see also §13.2).

[42] Exceptions are cubes and any other particles, for which area of any cross section perpendicular to z-axis is constant.

[43] That is additionally to the communication overhead that always exists (§13.3).

erroneous results. It may be useful if the scattering problem is symmetric, but **ADDA** do not recognize it automatically, e.g. for particles that are read from file or when not-default incident beam is used, which does not spoil the symmetry of the problem (e.g. plane wave propagating along the x-axis for a cubical scatterer). It is important to note that not the scatterer but its dipole representation should be symmetric,[44] otherwise the accuracy of the result will generally be slightly worse than that when symmetry is not used.

Particle symmetries can also be used to decrease the range of orientation/scattering angles for different averagings/integrations. However, it is user's responsibility to decide how a particular symmetry can be employed. This is described in the descriptions of corresponding input parameters files (§B.2, §B.3, §B.4).

9 Orientation of the Scatterer

9.1 Single orientation

Any particle orientation with respect to the laboratory reference frame can be specified by three Euler angles (α,β,γ). We use a notation based on [29], which also is called "*zyz*-notation" or "*y*-convention". In short, coordinate axes attached to the particle are first rotated by angle α over the z-axis, then by angle β over the current position of the y-axis (line of nodes), and finally by angle γ over the new position of the z-axis. These angles are specified in degrees as three arguments to the command line option

```
-orient <alpha> <beta> <gamma>
```

ADDA simulates light scattering in the particle reference frame (§8.1), therefore rotation of the particle is equivalently represented as an inverse rotation of the incident wave propagation direction and polarization (§10.1), position of the beam center (if relevant, §10.2), and scattering plane (angles). The information about the orientation of a scatterer is saved to the `log` (§C.4).

9.2 Orientation averaging

Orientation averaging is performed in **ADDA** over three Euler angles (α,β,γ). Rotating over α is equivalent to rotating the scattering plane without changing the orientation of the scatterer relative to the incident radiation. Therefore, averaging over this orientation angle is done with a single computation of internal fields; additional computation time for each scattering plane is comparably small. Averaging over the other two Euler angles is done by independent DDA simulations (defining the orientation of the scatterer as described in §9.1). The averaging itself is performed using the Romberg integration (§13.5), parameters of the averaging are stored by default in file `avg_params.dat` (§B.2). Orientation averaging is enabled by the command line option

```
-orient avg [<filename>]
```

where `<filename>` is an optional argument that specifies a different file with parameters of the averaging. Integration points for β are spaced uniformly in values of $\cos\beta$. Currently only the Mueller matrix in one scattering plane (§12.1), C_{ext}, and C_{abs} (§12.2) are calculated when doing orientation averaging. We are currently working to include the asymmetry vector **g** (§12.2) in this list. For now, orientation averaging is incompatible with integration of scattering amplitude over the whole solid angle.

It also can not be used in combination with saving incident beam (§10), internal fields or dipole polarizations (§12.4), or with calculating scattering for a grid of angles (§12.1).

[44] For example, a sphere is symmetric for any incident direction, but the corresponding dipole set (Fig. 3) is only symmetric for incidence along a coordinate axis.

10 Incident Beam

This section describes how to specify the incident electric field. This field, calculated for each dipole, can be saved to file `IncBeam` (§C.8). To enable this functionality specify command line option

```
-store_beam
```

10.1 Propagation direction

The direction of propagation of the incident radiation is specified by the command line option

```
-prop <x> <y> <z>
```

where arguments are x, y, and z components of the propagation vector. Normalization (to the unity vector) is performed automatically by **ADDA**. By default vector (0,0,1) is used. Two incident polarizations are used by default: along the x and y axis.[45] Those are perpendicular ($\perp$) and parallel ($\parallel$) polarizations [30] respectively with respect to the default scattering plane (yz). These polarizations are transformed simultaneously with the propagation vector – all three are rotated by two spherical angles (θ,φ) so that (0,0,1) is transformed into the specified propagation vector. All the scattering angles are specified with respect to the incident wave reference frame (§8.1) based on the *new* propagation vector (z) and two *new* incident polarizations (x,y).[46]

The option `-prop` is cumulative with rotation of the particle (§9.1) because the latter is equivalent to the inverse rotation of incident wave and scattering angles. If after all transformations the propagation vector is not equal to the default (0,0,1), all the symmetries of the scatterer are cancelled (§8.7).

10.2 Beam type

ADDA supports not only the ideal plane wave incident radiation, but also several types of finite size beams. The choice is determined by the command line option

```
-beam <type> [<width> <x> <y> <z>]
```

where `<type>` is one of the `plane`, `lminus`, `davis3`, or `barton5`. All beam types except the default plane wave are approximate descriptions of the Gaussian beam. Four arguments specified in the command line specify width (w_0) and x, y, z coordinates of the center of the beam respectively (all in μm). The coordinates are specified in the laboratory reference plane (§8.1). `lminus` is the simplest approximation [31], `davis3` [32] and `barton5` [33] are correct up to the third and fifth order of the beam confinement factor ($s = 1/kw_0$) respectively. The latter is recommended for all calculations; others are left mainly for comparison purposes. However, for tightly focused beams even `barton5` may be not accurate enough. We plan to implement a rigorous description of Gaussian beam, as well as other beam shapes. Moreover, adding of a new beam is straightforward for anyone who is familiar with C programming language. The procedure is described in §F.2.

For all beam types we assume unity amplitude of the electric field in the focal point of the beam. Some or all of the particle symmetries (§8.7) are cancelled according to the coordinates of the beam center. The validation of **ADDA** for simulation of a Gaussian beam is shown in §0.

[45] We are currently working to include an option to specify arbitrary (possibly complex) incident polarization. It can be useful if only one particular polarization need to be simulated.

[46] For example, the default scattering plane (§12.1), yz-plane, will be the one based on the new propagation vector and new incident polarization, which corresponds to the y-polarization for the default incidence.

11 DDA Formulation

Since its introduction by Purcell and Pennypacker [2] DDA has been constantly developing, therefore there exist a number of different DDA formulations. For an extensive review of these formulations the reader is referred to [12]. Here we only summarize the main results. All formulations are equivalent to the solution of the linear system to determine unknown dipole polarizations $\mathbf{P}_i$

$$\mathbf{E}_i^{\text{inc}} = \overline{\boldsymbol{\alpha}}_i^{-1}\mathbf{P}_i - \sum_{j\neq i}\overline{\mathbf{G}}_{ij}\mathbf{P}_j\,, \tag{9}$$

where $\mathbf{E}_i^{\text{inc}}$ is the incident electric field, $\overline{\boldsymbol{\alpha}}_i$ is the dipole polarizability (self-term), $\overline{\mathbf{G}}_{ij}$ is the interaction term, and indices i and j enumerate the dipoles. For a plane wave incidence

$$\mathbf{E}^{\text{inc}}(\mathbf{r}) = \mathbf{e}^0\exp(\mathrm{i}\mathbf{k}\cdot\mathbf{r})\,, \tag{10}$$

where $\mathbf{k} = k\mathbf{a}$, $\mathbf{a}$ is the incident direction, and $|\mathrm{e}^0| = 1$. Other incident beams are discussed in §10.2. The (total) internal electric field $\mathbf{E}_i$ is the one present in a homogenous particle modeled by an array of dipoles, it should be distinguished from the exciting electric field $\mathbf{E}_i^{\text{exc}}$ which is a sum of $\mathbf{E}_i^{\text{inc}}$ and the field due to all other dipoles, but excluding the field of the dipole i itself. Both total and exciting electric field can be determined once the polarizations are known:

$$\mathbf{P}_i = \overline{\boldsymbol{\alpha}}_i\mathbf{E}_i^{\text{exc}} = V\chi_i\mathbf{E}_i\,, \tag{11}$$

where $V = d^3$ is the volume of a dipole and $\chi_i = (\varepsilon_i - 1)/4\pi$ is the susceptibility of the medium at the location of the dipole (ε_i – relative permittivity). All scattering quantities also are determined from known polarizations. Below we discuss different formulations for the polarization prescription (§11.1), interaction term (§11.2) and formulae to calculate scattering quantities (§11.3). **ADDA** incorporates some new theoretical improvements that we are developing ourselves. They are in the early research phase, therefore we do not give any description for them. However, you may try them at your own risk.

11.1 Polarizability prescription

A number of expressions for the polarizability are known [12]. **ADDA** can use five of them: Clausius-Mossotti (CM), radiative reaction correction (RR), lattice dispersion relation (LDR), corrected LDR (CLDR), and Filtered Coupled Dipoles (FCD); and the second order (SO) polarization prescription, which we are developing ourselves. CM polarizability is the basic one, given by

$$\alpha_i^{\text{CM}} = d^3\frac{3}{4\pi}\frac{\varepsilon_i - 1}{\varepsilon_i + 2}\,. \tag{12}$$

RR is a third-order (in kd) correction to CM:

$$\alpha^{\text{RR}} = \frac{\alpha^{\text{CM}}}{1 - (2/3)\mathrm{i}k^3\alpha^{\text{CM}}}\,. \tag{13}$$

LDR adds second-order corrections:

$$\alpha^{\text{LDR}} = \frac{\alpha^{\text{CM}}}{1 - \left(\alpha^{\text{CM}}/d^3\right)\left[\left(b_1^{\text{LDR}} + b_2^{\text{LDR}}m^2 + b_3^{\text{LDR}}m^2S\right)(kd)^2 + (2/3)\mathrm{i}(kd)^3\right]}, \tag{14}$$

$$b_1^{\text{LDR}} \approx 1.8915316\,,\ b_2^{\text{LDR}} \approx -0.1648469\,,\ b_3^{\text{LDR}} \approx 1.7700004\,, \tag{15}$$

$$S = \sum_{\mu}\left(a_\mu e_\mu^0\right)^2\,, \tag{16}$$

where μ denote vector components. LDR prescription can be averaged over all possible incident polarizations [5], resulting in

$$S = \frac{1}{2}\left(1 - \sum_{\mu} a_{\mu}^4\right). \tag{17}$$

Corrected LDR is independent on the incident polarization but leads to the diagonal polarizability tensor instead of scalar

$$\alpha_{\mu\nu}^{\mathrm{CLDR}} = \frac{\alpha^{\mathrm{CM}}\delta_{\mu\nu}}{1 - \left(\alpha^{\mathrm{CM}}/d^3\right)\left[\left(b_1^{\mathrm{LDR}} + b_2^{\mathrm{LDR}} m^2 + b_3^{\mathrm{LDR}} m^2 a_{\mu}^2\right)(kd)^2 + (2/3)\mathrm{i}(kd)^3\right]}, \tag{18}$$

where $\delta_{\mu\nu}$ is the Kronecker symbol. FCD polarizability is obtained from the value of filtered Green's tensor [Eq. (22)] for zero argument [22], leading to

$$\alpha^{\mathrm{FCD}} = \frac{\alpha^{\mathrm{CM}}}{1 - \left(\alpha^{\mathrm{CM}}/d^3\right)\left[(4/3)(kd)^2 + (2/3)(\mathrm{i} + \ln((\pi - kd)/(\pi + kd))/\pi)(kd)^3\right]}. \tag{19}$$

Naturally, Eq. (19) is applicable only when $kd < \pi$, i.e. dpl > 2. CM, RR, LDR, and FCD can be used together with anisotropic electric permittivity, given by a diagonal tensor $\bar{\boldsymbol{\varepsilon}}$. Polarizability is then also a diagonal tensor, calculated by the same formulae [Eqs. (12)–(14)] but separately for each component:

$$\alpha_{\mu\nu} = \delta_{\mu\nu}\alpha(\varepsilon_{\mu\mu}). \tag{20}$$

The choice of the polarization prescription is performed by command line option
`-pol <type> [<arg>]`
where `<type>` is one of the `cm`, `rrc`, `ldr`, `cldr`, `fcd`, `so`. `<arg>` is optional flag that can be only `avgpol` and only for LDR – it specifies that LDR polarizability should be averaged over incident polarizations. Default is LDR without averaging. It is important to understand that this is not the best option for all cases. Our experience shows that LDR may perform particularly badly (much worse than CM or RR) for very large refractive indices, then FCD (together with its interaction term, §11.2) becomes the best option.

11.2 Interaction term

A few formulations for the interaction term are known [12]. Currently, **ADDA** can use the simplest one (interaction of point dipoles), FCD (in other words, filtered Green's tensor), quasistatic version of FCD, and the second order (SO) formulation (the latter we are developing ourselves). The interaction of point dipoles is described by the Green's tensor:

$$\overline{\mathbf{G}}_{ij} = \overline{\mathbf{G}}(\mathbf{r}_i, \mathbf{r}_j) = \frac{\exp(\mathrm{i}kR)}{R}\left[k^2\left(\bar{\mathbf{I}} - \frac{\hat{R}\hat{R}}{R^2}\right) - \frac{1 - \mathrm{i}kR}{R^2}\left(\bar{\mathbf{I}} - 3\frac{\hat{R}\hat{R}}{R^2}\right)\right], \tag{21}$$

where $\mathbf{r}_i$ is the radius-vector of the dipole center, $\mathbf{R} = \mathbf{r}_j - \mathbf{r}_i$, $R = |\mathbf{R}|$, $\bar{\mathbf{I}}$ is the identity tensor, and $\hat{R}\hat{R}$ is a tensor defined as $\hat{R}\hat{R}_{\mu\nu} = R_\mu R_\nu$. The filtered Green's tensor is defined [22] as

$$\overline{\mathbf{G}}_{ij}^{\mathrm{FCD}} = \bar{\mathbf{I}}\left(k^2 g_{\mathrm{F}}(R) + \frac{g_{\mathrm{F}}'(R)}{R} + \frac{4\pi}{3}h_r(R)\right) + \frac{\hat{R}\hat{R}}{R^2}\left(g_{\mathrm{F}}''(R) - \frac{g_{\mathrm{F}}'(R)}{R}\right), \tag{22}$$

where h_r is filter impulse response:

$$h_r(R) = \frac{\sin(k_{\mathrm{F}}R) - k_{\mathrm{F}}R\cos(k_{\mathrm{F}}R)}{2\pi^2 R^3}, \tag{23}$$

$k_{\mathrm{F}} = \pi/d$ – wavenumber corresponding to the grid, and g_{F} is filtered scalar Green's function:

$$g_{\mathrm{F}}(R) = \frac{1}{\pi R}\left\{\sin(kR)[\pi\mathrm{i} + \mathrm{Ci}((k_{\mathrm{F}} - k)R) - \mathrm{Ci}((k_{\mathrm{F}} + k)R)] + \cos(kR)[\mathrm{Si}((k_{\mathrm{F}} + k)R) + \mathrm{Si}((k_{\mathrm{F}} - k)R)]\right\}. \tag{24}$$

For applicability of this formulation k_F must be larger than k, i.e. dpl > 2. Quasistatic FCD is obtained in the limit $kR \to 0$, which leads to a simpler expression [34]:

$$\overline{\mathbf{G}}_{ij}^{\mathrm{FCD,st}} = -\frac{2}{3\pi R^3}\left(\bar{\mathbf{I}} - 3\frac{\hat{R}\hat{R}}{R^2}\right)\left[3\,\mathrm{Si}(k_F R) + k_F R\cos(k_F R) - 4\sin(k_F R)\right]. \tag{25}$$

Although it is just a special case of full FCD, it is implemented in **ADDA** as a separate option for testing and research purposes. However, it should be used only when scatterer is much smaller than the wavelength, otherwise it will lead to erroneous results. FCD was originally designed for high refractive indices, and we do recommend using it in this regime. The choice of the interaction term is performed by the command line option

```
-int <type>
```

where `<type>` is one of the `poi`, `fcd`, `fcd_st`, `so`. For SO formulation tables of precalculated integrals are used, they are automatically read from files in `tables/` (§D.1).

11.3 How to calculate scattering quantities

The simplest way to calculate scattering quantities is to consider a set of point dipoles with known polarizations, as summarized by Draine [3]. The scattering amplitude **F** for any scattering direction **n** is given as

$$\mathbf{F}(\mathbf{n}) = -\mathrm{i}k^3(\bar{\mathbf{I}} - \hat{n}\hat{n})\sum_i \mathbf{P}_i \exp(-\mathrm{i}k\mathbf{r}_i\cdot\mathbf{n}). \tag{26}$$

The Mueller scattering matrix for direction **n** is determined from **F**(**n**) calculated for two incident polarizations [30]. Scattering cross section C_{sca} and asymmetry vector **g** is determined by integration of **F**(**n**) over the whole solid angle:

$$C_{\mathrm{sca}} = \frac{1}{k^2}\oint \mathrm{d}\Omega\left|\mathbf{F}(\mathbf{n})\right|^2, \tag{27}$$

$$\mathbf{g} = \frac{1}{k^2 C_{\mathrm{sca}}}\oint \mathrm{d}\Omega\,\mathbf{n}\left|\mathbf{F}(\mathbf{n})\right|^2. \tag{28}$$

Extinction and absorption cross section (C_{ext} and C_{abs}) are determined directly from $\mathbf{P}_i$:

$$C_{\mathrm{ext}} = 4\pi k\sum_i \mathrm{Im}\left(\mathbf{P}_i\cdot\mathbf{E}_i^{\mathrm{inc}*}\right), \tag{29}$$

$$C_{\mathrm{abs}} = 4\pi k\sum_i [\mathrm{Im}(\mathbf{P}_i\cdot\mathbf{E}_i^{\mathrm{exc}*}) - (2/3)k^3|\mathbf{P}_i|^2]. \tag{30}$$

Variations of Eq. (30) (which is used by default) are possible, for instance [11]:

$$C_{\mathrm{abs}} = 4\pi k\sum_i \mathrm{Im}(\mathbf{P}_i\cdot\mathbf{E}_i^*), \tag{31}$$

which is based on considering radiation correction of a *finite* dipole instead of a *point* dipole used in Eq. (30). There is no difference between these two expressions for CM, RR, FCD polarizability formulations (§11.1) nor for real refractive index [12].

The choice between different ways to calculate scattering quantities is performed by command line option

```
-scat <type>
```

where `<type>` is `dr`, `fin`, or `so`. Classical Draine's formulation (`dr`) corresponds to Eqs. (26)-(30). Finite dipole correction (`fin`) uses Eq. (31) to calculate C_{abs}, and C_{ext} is obtained as Eq. (29) + Eq. (31) − Eq. (30). In other words, C_{ext} is corrected by the same amount as C_{abs} to ensure exact satisfaction of the optical theorem, which is discussed below. Second order formulation (`so`) is currently under development, so we do not provide any details about it here. A few quick tests showed that finite dipole correction do *not* improve the accuracy, so using the default `dr` is recommended in all cases. However, extensive testing is required to make any definite conclusions about the `fin` option.

C_{sca} can be determined as $C_{ext} - C_{abs}$, which is faster than Eq. (27). However, this issue needs further clarifying. Draine noted [3] that C_{sca} calculated by integration over the solid angle can be more accurate than $C_{ext} - C_{abs}$, when the latter two cross sections have close values. Moreover, it has been suggested [35] that difference between C_{sca} calculated by these two methods can be used as an internal measure of DDA accuracy. However, this difference is a measure of convergence of the iterative solver and accuracy of the integration over the solid angle, but not of the physical approximation itself. In other words, the difference may be very small, while the values themselves are very inaccurate (compared to exact solution). To prove it, one may consider a supplementary particle consisting of the (really) point dipoles with the same positions and polarizations as dipoles representing the initial particle. For this supplementary particle DDA equations (9) involve no physical approximation, and hence their exact solution will satisfy the optical theorem, which can be formulated as $C_{sca} = C_{ext} - C_{abs}$. Hence, the only possible reasons for the latter to be violated are inaccurate solution of Eq. (9) and inaccurate calculation of Eq. (27). The simulations agree with this conclusion (data not shown except for a particular example i°n parts 12 and 13 of the tutorial, see §G).

12 What Scattering Quantities Are Calculated

All the scattering angles (polar θ and azimuthal φ) are specified with respect to the incident wave (see §9.1 and §10.1 for details).

12.1 Mueller matrix

ADDA calculates a complete Mueller matrix, which is defined as in [30] without any normalization, for a set of scattering angles. By default scattering in the *yz*-plane is calculated. The range of [0°,180°] is equally divided into N_θ intervals. If the particle is not symmetric (§8.7) and orientation averaging (§9.2) is not used, the range is extended to 360 degrees. Totally $N_\theta + 1$ or $2N_\theta + 1$ points are calculated. N_θ is specified as an `<arg>` in command line option

```
-ntheta <arg>
```

By default N_θ is from 90 to 720 depending on the size of the computational grid (§8.2). To calculate the Mueller matrix in one scattering plane **ADDA** simulates two incident polarizations, however one is enough if the particle is symmetric with respect to the rotation by 90° over the propagation vector of incident radiation (§8.7).

More advanced options are available to calculate scattering at any set of angles. If any of the two command line options

```
-store_scat_grid
-phi_integr <arg>
```

is specified, the Mueller matrix is calculated for a set of angles, that are by default specified in a file `scat_params.dat` (§B.4). The first flag indicates that values of the Mueller matrix for all calculated angles should be saved to file `mueller_scatgrid` (§C.5), while the second flag turns on the integration of Mueller matrix over φ. `<arg>` is an integer from 1 to 31, each bit of which, from lowest to highest, indicates whether the integration should be performed with multipliers 1, $\cos(2\varphi)$, $\sin(2\varphi)$, $\cos(4\varphi)$, and $\sin(4\varphi)$ respectively.[47] Results of the integrations with multipliers specified by the `<arg>` are saved to files `mueller_integr`, `mueller_integr_c2`, `mueller_integr_s2`, `mueller_integr_c4`, and

[47] For example 1 corresponds to one integration with no multipliers, 6 – to two integration with $\cos(2\varphi)$ and $\sin(2\varphi)$ multipliers. Integration over φ with such multipliers is implemented because it appears in formulae for the light scattering patterns measured by the scanning flow cytometer [55,54], however they hopefully may also be useful in other applications.

`mueller_integr_s4` respectively (§C.5). It is important to note that the results of the integration are divided by the width of the φ interval (2π by default), i.e. actually averaging over φ takes place. If both above-mentioned command line options are specified, both initial and integrated results are saved to hard disk.

The format of the input file is very flexible (see §B.4 for details) allowing using either values uniformly spaced in some interval or any set of values, which is explicitly specified, for θ and φ independently. Even an arbitrary set of (θ,φ) pairs can be used. However, if integration over φ is used, a set of φ values must comply with the Romberg integration (§13.5). A different file describing a set of angles can be used if specified as an argument to the command line option

```
-scat_grid_inp <filename>
```

When a grid of scattering angles is calculated (either for saving or integrating over φ) the scattering in *yz*-plane is by default *not* calculated. However, **ADDA** may be forced to calculate it by specifying command line option

```
-yz
```

The Mueller matrix contains the complete information about angle-resolved scattering for any incident polarization, but other quantities are also commonly used, e.g. differential scattering cross section. Depending on the incident polarization it can be calculated as:

$$\frac{\mathrm{d}C_{\mathrm{sca}}^{\mathrm{unpol}}}{\mathrm{d}\Omega} = \frac{S_{11}}{k^2}, \frac{\mathrm{d}C_{\mathrm{sca}}^{x,y}}{\mathrm{d}\Omega} = \frac{1}{k^2}\left[S_{11} \pm S_{12}\cos(2\varphi) \pm S_{13}\cos(2\varphi)\right], \tag{32}$$

where plus and minus correspond to incident polarization along *x*- and *y*-axis respectively, and φ is the azimuthal angle of the scattering direction (relative to *x*-axis). "unpol" denotes unpolarized incident light. Eq. (32) can be easily derived from formulae given in [30].

12.2 Integral scattering quantities

All the scattering quantities described in this section are saved to file `CrossSec` (§C.6). Different files are used for two incident polarizations and when doing orientation averaging: `CrossSec-X`, `CrossSec-Y`, and `CrossSec` respectively. **ADDA** always calculates C_{ext} and C_{abs} (together with corresponding efficiencies Q_{ext}, Q_{abs}) and it can optionally calculate scattering cross section C_{sca} (and efficiency Q_{sca}) and normalized and not-normalized asymmetry vectors – $\mathbf{g}$ and $\mathbf{g}C_{\mathrm{sca}}$ respectively (the *z*-component of $\mathbf{g}$ is the usual asymmetry parameter $\langle\cos\theta\rangle$). Values of cross sections are in units of $\mu\mathrm{m}^2$. All the efficiencies are calculated by dividing the corresponding cross section over the area of the geometrical cross section of the sphere with volume equal to that of dipole representation of the particle. The optional features are turned on by command line options

```
-Csca
-vec
-asym
```

for calculation of C_{sca}, $\mathbf{g}C_{\mathrm{sca}}$ and $\mathbf{g}$ respectively. If $\mathbf{g}$ is calculated C_{sca} and $\mathbf{g}C_{\mathrm{sca}}$ are also calculated automatically.[48] The calculation of $\mathbf{g}$ and C_{sca} is performed by integration over the whole solid angle, the grid of scattering angles is used for it. The grid is specified by default in file `alldir_params.dat` (see §B.3 for format) in a form suitable for the Romberg integration (§13.5). Integration points for θ are spaced uniformly in values of $\cos\theta$. Different file describing the grid can be used if specified as an argument to the command line option

```
-alldir_inp <filename>
```

In most cases C_{sca} can be accurately calculated as $C_{\mathrm{ext}} - C_{\mathrm{abs}}$, without using `-Csca` option. As discussed in §11.3 accuracy of this method can be improved when $C_{\mathrm{sca}} \ll C_{\mathrm{abs}}$ by

[48] i.e. `-asym` implies `-Csca` and `-vec`.

using smaller ε (§13.1). Whether it is more computationally effective than using `-Csca` or not depends on the particular problem. For instance, C_{ext} and C_{abs} may coincide in (almost) all shown digits for particles much smaller than the wavelength, leaving integration of scattered fields over the solid angle as the only viable option. However, in this case the integration can be done analytically, using formulae for the Rayleigh regime. For a general non-symmetric particle formulae are cumbersome. However, if total particle polarizability is a diagonal tensor in the laboratory reference frame (e.g. general ellipsoid with axes along coordinate axes), then starting from chapter 5.7 of [30] one can derive:

$$C_{\text{sca}}^{x,y} = \frac{8\pi}{3}\left(S_{11} \mp S_{12}\right)\Big|_{\theta=0,\ \varphi=\pi/2}, \tag{33}$$

where plus and minus correspond to incident polarization along *y*- and *x*-axis respectively. Mueller matrix elements are considered at the forward direction in *yz*-plane, i.e. they can be found in the second line of the `mueller` file (§12.1).

12.3 Radiation forces

Radiation force for the whole scatterer and for each dipole can be calculated by **ADDA**. If the command line option

```
-Cpr_mat
```

is specified, the radiation force and efficiency vector are calculated and saved into file `CrossSec` (§C.6). If *additionally* an option

```
-store_force
```

is specified, the radiation forces on each dipole is saved into file `VisFrp` (§C.7). These features are still under development. More information can be found in a paper by Hoekstra *et al.* [16].

12.4 Internal fields and dipole polarizations

ADDA can save internal electric fields $\mathbf{E}_i$ and/or dipole polarizations $\mathbf{P}_i$ at each dipole (§11). They are saved to file `IntField` and `DipPol` respectively (§C.8). To enable this functionality specify one of or both command line options

```
-store_int_field
-store_dip_pol
```

These options are mainly for graphical representation and/or analysis of the internal fields and dipole polarizations, as well as for input into other software. To save internal fields for future use by **ADDA** consider using checkpoints of type "always" (§13.4). The accuracy of the internal fields in DDA simulations was studied by Hoekstra *et al.* [20].

12.5 Near-field

Near-field is the electric field near the particle, i.e. neither in the particle itself nor far enough to be considered a scattered field. Currently, **ADDA** can not calculate the near-field in a convenient manner. We are working on this feature; for now there are two possible workarounds. First, one may save the dipole polarizations to a file (§12.4) and perform the calculation of the electric field himself:

$$\mathbf{E}(\mathbf{r}) = \mathbf{E}^{\text{inc}}(\mathbf{r}) + \sum_i \overline{\mathbf{G}}(\mathbf{r}, \mathbf{r}_i)\mathbf{P}_i, \tag{34}$$

where $\overline{\mathbf{G}}$ is defined by Eq. (21). This workaround requires some programming from the user. Moreover, it may become computationally prohibitive, if near-field needs to be calculated at a very large number of points. Those who manage to write a program to calculate near-field at a set of points, which coordinates are read from file, are encouraged to send it to the authors, so other users may make use of it.

Second, one may include all the points, where near-field need to be calculated, in the particle geometry, but set their refractive index to a value very close to 1 (e.g. 1.00001). Then the scattering problem is identical to the initial one. Saving the internal fields (§12.4) will directly produce the required near-field together with the field inside the particle. The drawback of this workaround is increase of computational time, which depends on the size of the box containing the extended particle geometry. Moreover, a user is required to specify a shape file, except for the simplest cases. For instance, predefined shapes "`coated`" or "`spherebox`" (§8.4) can be used to calculate near-field in a sphere or cube around the homogenous sphere, respectively.

13 Computational Issues

13.1 Iterative solver

Main calculation part of a DDA simulation is finding a solution of a large system of linear equations; an iterative method is used for this purpose. **ADDA** incorporates 4 different methods: conjugate gradient applied to normalized equations with minimization of the residual norm (CGNR) [36], Bi-conjugate gradient (Bi-CG) [37,38], Bi-CG stabilized (Bi-CGStab) [36] and quasi minimal residual (QMR) [37]. Bi-CG and QMR employ the complex symmetric property of DDA interaction matrix to reduce the number of matrix-vector products per iteration by a factor of two [37].

The linear system, which is actually solved in **ADDA**, is in the form that is equivalent to the one that is Jacobi-preconditioned but kept complex-symmetric. Our experience suggests that QMR is generally the most efficient iterative solvers, however Bi-CGStab is faster in some cases. Performance of Bi-CG is comparable to that of QMR, but its convergence behavior is erratic, compared to that of Bi-CGStab. However, Bi-CG may become preferable when convergence is very slow and round-off errors significantly worsen the performance of QMR. CGNR is the slowest of the four, however it is very simple and its convergence is guaranteed to be monotonic [36]. QMR and BiCGSTAB require about 20% more RAM (for additional intermediate vectors) than CGNR and Bi-CG. Hence, Bi-CG may be also preferential when memory is sparse.

The iterative solver is chosen by the command line option

```
-iter <type>
```

where `<type>` is one of: `cgnr`, `bicg`, `bicgstab`, `qmr`. By default QMR is used. The initial vector is automatically chosen by **ADDA** from two variants: zero or the incident field. The former is completely general, while the latter may be closer to the exact solution for small index-matching particles. **ADDA** chooses the variant that gives lesser residual norm,[49] this choice is shown in the `log` (§C.4).

The stopping criterion for iterative solvers is relative norm of the residual – the process stops when this norm is less than ε. The latter can be specified by the command line option

```
-eps <arg>
```

where $\varepsilon = 10^{-\langle arg \rangle}$. By default $\varepsilon = 10^{-5}$. The maximum number of iterations can be specified as `<arg>` to the command line option

```
-maxiter <arg>
```

ADDA will stop execution if the iterative solver does not converge in the given number of iterations. By default the maximum number of iterations is set to a very high value, which is not expected to be ever reached.[50] **ADDA** will also stop if the iterative solver stagnates for a

[49] It should be noted, however, that smaller residual of the initial vector does not necessarily leads to a faster convergence [36].

[50] Currently it is set to $3N$, i.e. the number of equations in a linear system.

long time, i.e. residual norm do not decrease during a number of last iterations. This number is currently set to 50000 for QMR and Bi-CG, 30000 for BiCGSTAB, and 10 for CGNR.[51]

All iterative solvers except CGNR are susceptible to breakdowns. Although such situations are very improbable, they may happen in some specific cases. For instance, simulation of light scattering by cube with edge length that is exactly a multiple of the wavelength results in breakdowns of QMR and Bi-CG. **ADDA** should detect almost any breakdown and produce a meaningful error message; however, it is possible that some breakdown is not detected. It then results in the stagnation of the residual norm or its oscillations without any further progress. On contrary, **ADDA** may claim breakdown when the convergence is slow but still satisfactory. The behavior of breakdown detection may be modified,[52] but it requires recompiling the code.

If you notice any error of the breakdown detection, both false-negative and false-positive, please communicate it to the authors. If the breakdown does appear, the solution is either to try different iterative solver or adjust input parameters a little bit. Usually changing particle size (or wavelength) by only 0.01% completely eliminates the breakdown.

13.2 Fast Fourier transform

The iterative method to solve a system of linear equations that arouse in DDA accesses the interaction matrix only by the means of calculating matrix-vector products. This can be done in O(Nln$N$) operations (N – total number of dipoles) using the FFT [39]. 3D (parallel) FFT is used in **ADDA**, however it is explicitly decomposed into a set of 1D FFTs, which allows reduction of calculations since only part of the array, on which FFT is performed, is actually used (see [15] for details).

1D FFTs are performed using standard libraries – two are implemented in **ADDA**: a routine by Temperton (`CFFT99`, [40]), which is included in the code, or the more advanced package FFTW 3 [41]. The latter is generally significantly faster, but requires separate installation of the package (§4). The FFT routine to use is chosen at compile time. By default FFTW 3 is used; to use Temperton's routine uncomment the line

```
CFLAGS += -DFFT_TEMPERTON
```

in `Makefile` and recompile (see §4).

FFT is performed on the grid that is doubled in each dimension compared to the computational grid. Temperton's FFT requires that the dimensions of this grid be of the form $2^p3^q5^r$ (all exponents are integers), FFTW 3 works with any grid dimensions but is most effective for dimensions of the form $2^p3^q5^r7^s11^u13^v$ ($u+v\le 1$). It should not be a limitation for the sequential mode, since **ADDA** automatically increases the FFT-grid dimensions to the first number of the required form.[53] But in parallel mode these dimensions must also divide the number of processors. Therefore the increase of the dimensions (and hence simulation time) may be substantial, and not possible at all if the number of processors divide any prime number larger than 5 for Temperton FFT or has more than one divisor of either 11 or 13 (or at least one larger than 13) for FFTW 3. Therefore it is strongly recommended *not* to use such "weird" number of processors.[54] It is the user's responsibility to optimize the combination of computational grid dimensions and number of processors, although **ADDA** will work, but probably not efficiently, for most of the combinations (see also §0).

Symmetry of the DDA interaction matrix is used in **ADDA** to reduce the storage space for the Fourier-transformed matrix, except when SO formulae to calculate interaction term are used (§11.2). This option can be disabled (mainly for debugging purposes) by specifying

[51] They are defined in lines starting with "`#define MAXCOUNT_`" in the beginning of the `iterative.c`.
[52] Changing values in lines starting with "`#define EPS_`" in the beginning of the `iterative.c`.
[53] The maximum increase is 15% for Temperton FFT and 6% for FFTW 3.
[54] Otherwise Temperton FFT will fail and FFTW 3 will perform less efficiently.

```
-no_reduced_fft
```
in the command line.

13.3 Parallel performance

ADDA is capable of running on a multiprocessor system, parallelizing a single DDA simulation. It uses MPI for communication routines. The principal limitation of DDA simulations on a desktop system is the amount of RAM available. For **ADDA** this limitation only specifies the minimum number of nodes (with separate memory for each node) to use. More nodes can be used to accelerate the computations. However, the more nodes assigned – the more simulation time (relative to the total time) is spent on communications. One should also take into account that when many nodes (processors) are used the MPI interface may occupy significant amount of RAM on each node, thereby decreasing the RAM available for **ADDA** itself (see also §7 and §0). This depends on the specific MPI implementation and/or hardware.

13.4 Checkpoints

ADDA is capable of creating checkpoints, in which the complete running state is saved and can be restored afterwards. All the intermediate vectors of the iterative solver (§13.1) are saved. This allows restarting the iterative solver exactly at the position, where the checkpoint was created. Time of a checkpoint is specified by command line option
```
-chpoint <time>
```
where `<time>` is time in format "`#d#h#m#s`".[55] There are 3 possible strategies for checkpoints, which are specified by the command line option
```
-chp_type <type>
```
where `<type>` is one of `normal`, `regular`, `always`. "Normal" means that after the checkpoint time elapsed the checkpoint is saved as soon as possible (it waits for the finishing of the current iteration) and **ADDA** finishes execution without any further actions. This type is useful when one need **ADDA** to run not longer than certain time. "Regular" checkpoints are saved after every specified time interval but do not influence the normal execution of **ADDA** – it runs until simulation is fully completed. Use this option when abrupt termination of **ADDA** may occur (e.g. system crash or the system resources are urgently needed for other tasks). "Always" type is similar to "normal" if checkpoint time elapsed during the execution, however it will also save a checkpoint (after the last iteration) when **ADDA** finishes normally earlier. That is the only checkpoint type, for which time may be not specified (then it is equivalent to infinite time). It may be useful if the simulation is valuable by itself but may be extended in the future, e.g. to obtain better convergence (lower ε, §13.1) or to calculate different scattering quantities (§12).

To restart the simulation from a checkpoint specify in the command line
```
-chp_load
```
The user should take care that the simulation is restarted with the same parameters that were used when saving the checkpoint. Although some parameters can indeed be different (e.g. those determining the output of **ADDA**), the consistency of the results is user's responsibility. By default all the checkpoint data is saved in the directory `chpoint` (§D.2), however a different directory (the same for saving and loading of checkpoints) can be specified as an argument to the command line option
```
-chp_dir <dirname>
```
The total size of the checkpoint files is approximately half of the RAM used, therefore 1) enough space on the hard disk should be available; 2) saving of a checkpoint may take

[55] All fields are optional, "`s`" can be omitted, the format is not case sensitive. For example: "`12h30M`", "`1D10s`", "`3600`" (equals 1 hour).

considerable time. Both issues are especially relevant for large simulations on multiprocessor systems. If the simulation time is strictly limited (e.g. by a batch system of a supercomputer with shared usage) checkpoint time should be set slightly smaller, so that **ADDA** would have enough time to finish the iteration and save a checkpoint (and possibly to calculate the scattering quantities if the iterative solver will converge just before the checkpoint time). User should estimate the needed time reserve himself. When loading a checkpoint, **ADDA** initializes anew, this takes some time. However, this time is usually small compared to the time used for the iterations.

It is also important to note that by default the same checkpoint directory is used for all the simulations on the current system that are run from the same path, therefore new checkpoints overwrite the old ones.[56] To avoid it specify a different checkpoint directory for each instance of **ADDA**; it is obligatory when several instances of **ADDA** run in parallel. For now, **ADDA** *always* saves checkpoints into the same directory where it loads it from.

Currently only the state of the iterative solver is saved to checkpoint, therefore it is suitable only for a simulation for a single incident polarization. We are working to extend its applicability to standard non-symmetric particles (two incident polarizations) and to orientation averaging.

13.5 Romberg integration

Integration is performed in **ADDA** for different purposes: orientation averaging (§9.2), integration of the Mueller matrix over the azimuthal angle (§12.1), and integration of the scattered field over the whole solid angle (§12.2). The same routine is used for all these purposes, which is based on the one- or two-dimensional Romberg integration [42]. It is a high-order technique that may be used in adaptive mode (it automatically performs only the necessary number of function evaluations to reach a prescribed accuracy). Adaptability is relevant for orientation averaging, where each function evaluation is a complete DDA simulation, but not for integration over scattering angles, because in this case all the values are precalculated. The Romberg integration also provides an estimate of the integration error, which is reliable for "sufficiently nice" functions [42]. When 2D integration is performed **ADDA** integrates an error estimate obtained in the inner integration loop simultaneously with the function values. It allows to obtain reliable estimate of the final error in the outer loop. The information about the integration together with errors is saved to separate log files (§C.9): `log_orient_avg`, `log_int_Csca-X`, `log_int_asym_x-X`, `log_int_asym_y-X`, `log_int_asym_z-X` for orientation averaging, calculation of C_{sca} and each component of $\mathbf{g}$ respectively (the last 4 file names have different suffixes – `X` or `Y` – for different incident polarizations). For orientation averaging some information is saved to the main `log` (§C.4). For integration of the Mueller matrix over the azimuthal angle only the averaged errors are saved together with the values directly to `mueller_integr` files (§C.5)

The drawback of the Romberg integration is that argument values must be uniformly spaced and their total number is limited to be $2^n + 1$ (n is any integer). The set of integration points is specified by minimum and maximum values and minimum and maximum number of subdivisions (refinements) $J_{\min}$ and $J_{\max}$ (the latter is equal to n). The required accuracy to reach is also a parameter. In some cases minimum and maximum values of some arguments are equivalent (e.g. 0° and 360° for φ), **ADDA** accounts for it to slightly decrease simulation time.

If the function is "nice" and periodic over the interval of integration, then it is not beneficial and may be even harmful to use higher order integration scheme. For such cases the simplest trapezoid rule (or equivalently midpoint rule) is the most efficient [42]. The same is

[56] This is done so to save the hard disk space.

true, when e.g. the function is integrated over the half of its period and is symmetric with respect to both borders of the integration interval. **ADDA**'s behavior is determined by the flag `periodic` in the input files. If it is `true` then trapezoid rule is used as a special case of the general Romberg scheme. This keeps the limitation on the number of integration points, but provides adaptability and reliable error estimates. All the integration parameters are specified in input files: `avg_params.dat` (§B.2), `scat_params.dat` (§B.4), `alldir_params.dat` (§B.3) corresponding to different integration tasks.

14 Timing

14.1 Basic timing

The basic timing of **ADDA** execution on a single processor is performed using standard ANSI C functions `clock` and `time`, which are completely portable. The drawbacks are low precision (1 s) of wall-time and low precision (0.1 s on most systems) and possible overflows (after 1 hour on 32-bit systems) of the processor timer. **ADDA** uses wall-time only for the total execution time and timing of checkpoints (§13.4), and for the rest processor time is measured. It makes timing more precise especially on desktop computers that are partly occupied by other tasks; however, for long simulations some timing results may become meaningless because of the timer overflow. The problem with `clock` overflows should not be relevant in 64-bit environment (§4.1) because of larger size of `clock_t` type.

In parallel mode all the times are wall-times measured by the high-precision `MPI_Wtime` function, which is a part of MPI standard. This solves above-mentioned problems of the ANSI C timers.

Timing results are presented in the end of the `log` (§C.4) together with some statistics (total number of iterations, total number of planes where the scattered field is calculated). Timing covers all the major parts: initialization (including initialization of FFT, §13.2, building the interaction matrix, and constructing a dipole representation of the particle, §8.3), solution for the internal fields (including iterative solver, §13.1), calculation of the scattering quantities (scattered electric field and others, §12), input/output (including checkpoint loading/saving, §13.4), integration (§13.5). Some are divided into subsections. If `-prognosis` option is used, only the time for constructing a particle is shown.

In parallel mode communication time (between different processors) is shown separately for some tasks. To measure this time accurately synchronization is done before the communication takes place. Our experience shows that this should not decrease the performance of **ADDA**, except for the small effect in the granule generator (§8.5).

14.2 Precise timing

This feature of **ADDA** is used to perform the thorough timing of the most computationally intensive parts: initialization of interaction matrix and FFT (especially FFTW 3, §13.2) and matrix-vector product. It gives the detailed information both on FFT and algebraic parts, which can be used for deep optimization or performance studies. However, this regime is incompatible with the normal **ADDA** execution – it terminates after the first matrix-vector product. Only the `stdout` is produced (§C.2). Precision of the precise timing routines are of order μs, however they measure wall-time[57] and are operating-system-dependent. The latter should not be a big problem, since **ADDA** contains routines for any POSIX or Windows

[57] It is hard to implement routines to measure processor time with better precision than that of standard `clock`, since all such routines are processor dependent.

operating systems; the appropriate one is automatically chosen by compiler directives.[58] To enable precise timing uncomment the line

```
CFLAGS += -DPRECISE_TIMING
```

in `Makefile` and recompile (see §7).

15 Miscellanea

Additional files contributed by users will be located in the directory `misc/`. These files should be considered to be *not supported* by the authors. They should be accompanied by enough information to explain their use.

16 Finale

This manual is somewhat inelegant, but we hope that it will prove useful. The structure of the input files is intended to be simple and suggestive so that, after reading the above notes once, the users may not have to refer to them again. Command line help system (`-h` option, §5.1) is intended to provide quick reference on different command line options. If that is not enough, thorough description of all command line options can be found in this manual (§A).

Users are encouraged to subscribe to `adda-announce` Google group[4]; bug reports and any new releases of **ADDA** will be made known to those who do. All known bugs are available at issue tracker,[2] it also lists the future plan for **ADDA** development. Users may "star" issues, which they consider important, to influence their priorities (Google account is required for that). Specific suggestions for improving **ADDA** and this manual are welcomed. You may either submit it directly to the issue tracker[2] or discuss it in the **ADDA** discussion group.[5]

If you have a question about **ADDA**, which you think is common, please look into the `doc/faq` file before searching for the answer in this manual. If neither of these helps, direct your questions to `adda-discuss` Google group.[59] The archive of this group is available,[60] so you may try to find an answer to your question before posting it. However, this also means that you should *not* include (potentially) confidential information in your message; in that case contact the authors directly. We also encourage users interested in different aspects of **ADDA** usage to join this group to participate in discussions initiated by other people.[61]

Finally, the authors have one special request: we would very much appreciate preprints and (especially!) reprints of any papers which make use of **ADDA**.

17 Acknowledgements

- The CFFT99 Fortran routine was written by Clive Temperton (1980).
- The FFTW 3 package, to which we link, was written by Matteo Frigo and Steven G. Johnson (`fftw@fftw.org`).
- The Mersenne twister random number generator was written by Makoto Matsumoto and Takuji Nishimura.
- The FFT part of **ADDA** (`fft.c`), matrix-vector product (`matvec.c`), and most of the non-standard beam types (`GenerateB.c`) were first implemented by Michel D. Grimminck. He also contributed to the particle generation routines (`make_particle.c`)

[58] Timing routines for some other operating system may be implemented instead of the current ones in source files `prec_timing.c` and `prec_timing.h`.
[59] Just send an e-mail to `adda-discuss@googlegroups.com`
[60] `http://groups.google.com/group/adda-discuss/topics`
[61] To do this send an e-mail to `adda-discuss-subscribe@googlegroups.com`

- The MPI part of **ADDA** (`comm.c`), 2D Romberg integration (`Romberg.c`), and calculation of the radiation force and scattering quantities obtained by integration (`crosssec.c`) were first implemented by Martin Frijlink.
- Several new shapes were added to the older version of particle generation routine (`make_particle.c`) by Konstantin A. Semyanov.
- Egg and capsule shapes were implemented by Daniel Hahn and Richard Joseph.
- The axisymmetric shape was implemented based on the code by Konstantin Gilev.

We are deeply indebted to all of these authors for making their code available. We also thank:

- Bruce T. Draine and Piotr J. Flatau for creating and developing the first publicly available DDA code "DDSCAT". They maintain a high standard, which we try to match, both for highly productive convenient code and clear detailed manual. Parts of the user guide for DDSCAT 6.1 [26] were used when writing this manual.
- Roman Schuh and Thomas Wriedt for providing data on scattering of a Gaussian beam by a sphere.
- Alexander N. Shvalov for information on MinGW and encouraging the authors to produce executables for Windows.
- Michiel Min for fruitful discussions about the `-anisotr` option and his motivation to implement FCD formulation.
- R. Scott Brock for ideas to greatly accelerate computation of the scattered field.
- All **ADDA** users who provided bug reports and suggestions, especially Antti Penttila and Vitezslav Karasek.

Development of **ADDA** was supported by The University of Amsterdam (The Netherlands), The Dutch Science Foundation NWO (The Netherlands), The Department of Defense (USA), and The NATO Science for Peace program.

18 References

[1] F.M. Kahnert, "Numerical methods in electromagnetic scattering theory," *J.Quant.Spectrosc.Radiat.Transfer* **79**, 775-824 (2003).

[2] E.M. Purcell and C.R. Pennypacker, "Scattering and adsorption of light by nonspherical dielectric grains," *Astrophys.J.* **186**, 705-714 (1973).

[3] B.T. Draine, "The discrete dipole approximation and its application to interstellar graphite grains," *Astrophys.J.* **333**, 848-872 (1988).

[4] B.T. Draine and P.J. Flatau, "Discrete-dipole approximation for scattering calculations," *J.Opt.Soc.Am.A* **11**, 1491-1499 (1994).

[5] B.T. Draine and J.J. Goodman, "Beyond clausius-mossotti - wave-propagation on a polarizable point lattice and the discrete dipole approximation," *Astrophys.J.* **405**, 685-697 (1993).

[6] B.T. Draine, "The discrete dipole approximation for light scattering by irregular targets," *Light Scattering by Nonspherical Particles, Theory, Measurements, and Applications*, M.I. Mishchenko, J.W. Hovenier, and L.D. Travis, eds., New York: Academic Press, 2000, pp. 131-145.

[7] G.H. Goedecke and S.G. O'Brien, "Scattering by irregular inhomogeneous particles via the digitized Green's function algorithm," *Appl.Opt.* **27**, 2431-2438 (1988).

[8] A. Lakhtakia, "Strong and weak forms of the method of moments and the coupled dipole method for scattering of time-harmonic electromagnetic-fields," *Int.J.Mod.Phys.C* **3**, 583-603 (1992).

[9] J. Rahola, "Solution of dense systems of linear equations in the discrete-dipole approximation," *SIAM J.Sci.Comp.* **17**, 78-89 (1996).

[10] N.B. Piller, "Coupled-dipole approximation for high permittivity materials," *Opt.Comm.* **160**, 10-14 (1999).
[11] P. Chaumet, A. Sentenac, and A. Rahmani, "Coupled dipole method for scatterers with large permittivity," *Phys.Rev.E* **70**, 036606 (2004).
[12] M.A. Yurkin and A.G. Hoekstra, "The discrete dipole approximation: an overview and recent developments," *J.Quant.Spectrosc.Radiat.Transfer* **106**, 558-589 (2007).
[13] A.G. Hoekstra and P.M.A. Sloot, "New computational techniques to simulate light-scattering from arbitrary particles," *Part.Part.Sys.Charact.* **11**, 189-193 (1994).
[14] A.G. Hoekstra and P.M.A. Sloot, "Coupled dipole simulations of elastic light scattering on parallel systems," *Int.J.Mod.Phys.C* **6**, 663-679 (1995).
[15] A. Hoekstra, M. Grimminck, and P. Sloot, "Large scale simulations of elastic light scattering by a fast discrete dipole approximation," *Int.J.Mod.Phys.C* **9**, 87-102 (1998).
[16] A.G. Hoekstra, M. Frijlink, L.B.F.M. Waters, and P.M.A. Sloot, "Radiation forces in the discrete-dipole approximation," *J.Opt.Soc.Am.A* **18**, 1944-1953 (2001).
[17] M.A. Yurkin, V.P. Maltsev, and A.G. Hoekstra, "The discrete dipole approximation for simulation of light scattering by particles much larger than the wavelength," *J.Quant.Spectrosc.Radiat.Transfer* **106**, 546-557 (2007).
[18] S.B. Singham, "Theoretical factors in modeling polarized light scattering by arbitrary particles," *Appl.Opt.* **28**, 5058-5064 (1989).
[19] A.G. Hoekstra and P.M.A. Sloot, "Dipolar unit size in coupled-dipole calculations of the scattering matrix-elements," *Opt.Lett.* **18**, 1211-1213 (1993).
[20] A.G. Hoekstra, J. Rahola, and P.M.A. Sloot, "Accuracy of internal fields in volume integral equation simulations of light scattering," *Appl.Opt.* **37**, 8482-8497 (1998).
[21] I. Ayranci, R. Vaillon, and N. Selcuk, "Performance of discrete dipole approximation for prediction of amplitude and phase of electromagnetic scattering by particles," *J.Quant.Spectrosc.Radiat.Transfer* **103**, 83-101 (2007).
[22] N.B. Piller and O.J.F. Martin, "Increasing the performance of the coupled-dipole approximation: A spectral approach," *IEEE Trans.Ant.Propag.* **46**, 1126-1137 (1998).
[23] M. Yurkin, V. Maltsev, and A. Hoekstra, "Convergence of the discrete dipole approximation. II. An extrapolation technique to increase the accuracy," *J.Opt.Soc.Am.A* **23**, 2592-2601 (2006).
[24] O.J.F. Martin and N.B. Piller, "Electromagnetic scattering in polarizable backgrounds," *Phys.Rev.E* **58**, 3909-3915 (1998).
[25] R. Schmehl, B.M. Nebeker, and E.D. Hirleman, "Discrete-dipole approximation for scattering by features on surfaces by means of a two-dimensional fast Fourier transform technique," *J.Opt.Soc.Am.A* **14**, 3026-3036 (1997).
[26] B.T. Draine and P.J. Flatau, "User guide for the discrete dipole approximation code DDSCAT 6.1," *http://arxiv.org/abs/astro-ph/0409262* (2004).
[27] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans.Model.Comp.Simul.* **8**, 3-30 (1998).
[28] M. Yurkin, K. Semyanov, V. Maltsev, and A. Hoekstra, "Discrimination of granulocyte subtypes from light scattering: theoretical analysis using a granulated sphere model," *Opt.Express* **15**, 16561-16580 (2007).
[29] M.I. Mishchenko, "Calculation of the amplitude matrix for a nonspherical particle in a fixed orientation," *Appl.Opt.* **39**, 1026-1031 (2000).
[30] C.F. Bohren and D.R. Huffman, *Absorption and scattering of Light by Small Particles*, New York: Wiley, 1983.

[31] G. Gouesbet, B. Maheu, and G. Grehan, "Light-scattering from a sphere arbitrarily located in a Gaussian-beam, using a Bromwich formulation," *J.Opt.Soc.Am.A* **5**, 1427 - 1443 (1988).
[32] L.W. Davis, "Theory of electromagnetic beams," *Phys.Rev.A* **19**, 1177-1179 (1979).
[33] J.P. Barton and D.R. Alexander, "Fifth-order corrected electromagnetic-field components for a fundamental Gaussian-beam," *J.Appl.Phys.* **66**, 2800-2802 (1989).
[34] P. Gay-Balmaz and O.J.F. Martin, "A library for computing the filtered and non-filtered 3D Green's tensor associated with infinite homogeneous space and surfaces," *Comp.Phys.Comm.* **144**, 111-120 (2002).
[35] E.S. Zubko, H. Kimura, Y.G. Shkuratov, K. Muinonen, T. Yamamoto, and G. Videen, "Light scattering by agglomerated debris particles composed of highly absorbing material," *Proceedings of the 11th Conference on Electromagnetic and Light Scattering*, Hatfield, UK: 2008, pp. 213-216.
[36] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H.A. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, 1994.
[37] R.W. Freund, "Conjugate gradient-type methods for linear-systems with complex symmetrical coefficient matrices," *SIAM J.Sci.Stat.Comp.* **13**, 425-448 (1992).
[38] H.A. van der Vorst and J.B.M. Melissen, "A Petrov-Galerkin type method for solving Ax=b, where A is symmetric complex," *IEEE Trans.Magn.* **26**, 706-708 (1990).
[39] J.J. Goodman, B.T. Draine, and P.J. Flatau, "Application of fast-Fourier-transform techniques to the discrete-dipole approximation," *Opt.Lett.* **16**, 1198-1200 (1991).
[40] C. Temperton, "Self-sorting mixed-radix fast Fourier transforms," *J.Comp.Phys.* **52**, 1-23 (1983).
[41] M. Frigo and S.G. Johnson, "The design and implementation of FFTW3," *IEEE Proc.* **93**, 216-231 (2005).
[42] P.J. Davis and P. Rabinowitz, *Methods of Numerical Integration*, New York: Academic Press, 1975.
[43] A. Penttila, E. Zubko, K. Lumme, K. Muinonen, M.A. Yurkin, B.T. Draine, J. Rahola, A.G. Hoekstra, and Y. Shkuratov, "Comparison between discrete dipole implementations and exact techniques," *J.Quant.Spectrosc.Radiat.Transfer* **106**, 417-436 (2007).
[44] M. Yurkin, A. Hoekstra, R. Brock, and J. Lu, "Systematic comparison of the discrete dipole approximation and the finite difference time domain method for large dielectric scatterers," *Opt.Express* **15**, 17902-17911 (2007).
[45] INMOS Ltd., *Occam 2 Reference Manual*, Prentice Hall, 1988.
[46] A.G. Hoekstra, L.O. Hertzberger, and P.M.A. Sloot, "Simulation of elastic light scattering on distributed memory machines," *Bull. Am. Phys. Soc.* **36**, 1798 (1991).
[47] P.M.A. Sloot and A.G. Hoekstra, "Implementation of a parallel conjugate gradient method for simulation of elastic light scattering," *Computers in Physics* **6**, 323 (1992).
[48] P.M.A. Sloot and A.G. Hoekstra, "Light scattering simulations on a massively parallel computer at the IC3A," *Transput. Applic.* **1**, 42-49 (1993).
[49] A.G. Hoekstra, P.M.A. Sloot, F. van der Linden, M. van Muiswinkel, J.J.J. Vesseur, and L.O. Hertzberger, "Native and generic parallel programming environments on a transputer and a PowerPC platform," *Concurrency Pract. Ex.* **8**, 19-46 (1996).
[50] P.M.A. Sloot, A.G. Hoekstra, and L.O. Hertzberger, "A comparison of the Iserver-Occam, Parix, Express, and PVM programming environments on a Parsytec GCel," *High-Performance Computing and Networking: International Conference and Exhibition*, W. Gentzsch and U. Harms, eds., Berlin: Springer-Verlag, 1994, pp. 253-259.

[51] A.G. Hoekstra, P.J.H. Brinkhorst, and P.M.A. Sloot, "First results of DDA simulations of elastic light scattering by red blood cell," *Proceedings of Workshop on Light Scattering by Non-spherical Particles*, K. Lumme, J.W. Hovenier, K. Muinonen, J. Rahola, and H. Laitinen, eds., Helsinki: University of Helsinki, 1997, pp. 39-40.
[52] D.V. Hahn, D. Limsui, R.I. Joseph, K.C. Baldwin, N.T. Boggs, A.K. Carr, C.C. Carter, T.S. Han, and M.E. Thomas, "Shape characteristics of biological spores," *SPIE Proc.* **6954**, 69540W (2008).
[53] P.W. Kuchel and E.D. Fackerell, "Parametric-equation representation of biconcave erythrocytes," *Bull.Math.Biol.* **61**, 209-220 (1999).
[54] M.A. Yurkin, K.A. Semyanov, P.A. Tarasov, A.V. Chernyshev, A.G. Hoekstra, and V.P. Maltsev, "Experimental and theoretical study of light scattering by individual mature red blood cells with scanning flow cytometry and discrete dipole approximation," *Appl.Opt.* **44**, 5249-5256 (2005).
[55] V.P. Maltsev and K.A. Semyanov, *Characterisation of Bio-Particles from Light Scattering*, Utrecht: VSP, 2004.

A Command Line Options

Most of the parameters are specified to **ADDA** in the command line. **ADDA** will automatically detect most inconsistencies in input parameters, and produce an error or warning if necessary (§C.1). If you notice that particular command line option causes abrupt termination or any erroneous behavior of **ADDA**, please communicate it to the authors. Command line option `-h` can be used to get both general help and detailed description of other command line options.

Below is the full list of command line options in alphabetical order. "`<…>`" denotes an argument, "`[…]`" denotes an optional part, and "`{…|…}`" denotes different possibilities. "Default:" and "Example:" fields specify the values of parameters excluding the command line option itself. Default for all the flags (that can be used without arguments) is false, i.e. if the flag is not specified, the corresponding option is not enabled.

`-alldir_inp <filename>`
: Specifies a file with parameters of the grid of scattering angles for calculating integral scattering quantities (§12.2). Input format is described in §B.3.
Default: alldir_params.dat

`-anisotr`
: Specifies that refractive index is anisotropic (§8.3). Option `-m` then accepts 6 arguments per each domain. Can not be used with CLDR polarizability (§11.1) and all SO formulations (§11.1, §11.2, §11.3).

`-asym`
: Calculate the asymmetry vector (§12.2). Implies `-Csca` and `-vec`.

`-beam {plane|{lminus|davis3|barton5} <width> [<x> <y> <z>]}`
: Sets a type of the incident beam (§10.2). Four other float arguments are relevant for all beam types except the plane wave. These are the width and x, y, z coordinates of the center of the beam respectively (all in μm). The coordinates are specified in the reference plane of the beam relative to the center of the computational box. They can be omitted – then beam center is located in the origin.
Default: `plane`
Example: barton5 5 0.3 0 0

`-chp_dir <dirname>`
: Sets directory for the checkpoint (both for saving and loading, §13.4).
Default: `chpoint`

`-chp_load`
: Restart a simulation from a checkpoint (§13.4).

`-chp_type {normal|regular|always}`
: Sets type of the checkpoint (§13.4). All types, except `always`, require `-chpoint`.
Default: `normal`

`-chpoint <time>`
: Specifies the time for checkpoints (§13.4) in format "`#d#h#m#s`". All fields are optional, numbers are integers, "`s`" can be omitted, the format is not case sensitive.
Examples: `12h30M`, `1D10s`, `3600`

`-Cpr_mat`
: Calculate the total radiation force (§12.3).

`-Csca`
: Calculate scattering cross section (by integrating the scattered field, §12.2).

`-dir <dirname>`
Sets directory for output files.
Default: constructed automatically, see §C.3.

`-dpl <arg>`
Sets parameter "dipoles per lambda" (§8.2), float.
Default: $10|m|$, where m is the maximum (by absolute value) refractive index specified by the "`-m`" option (§8.3).

`-eps <arg>`
Specifies the stopping criterion for the iterative solver (§13.1) by setting the relative norm of the residual ε to reach. `<arg>` is an exponent of base 10 (float), i.e. $\varepsilon = 10^{-<\text{arg}>}$.
Default: `5` ($\varepsilon = 10^{-5}$)

`-eq_rad <arg>`
Sets volume-equivalent radius of the particle in μm (§8.2), float. If default wavelength is used, this option specifies the volume-equivalent size parameter. Can not be used together with `-size`. Size is defined by some shapes themselves, then this option can be used to override the internal specification and scale the shape.
Default: determined by the value of `-size` or by `-grid`, `-dpl`, and `-lambda`.

`-granul <vol_frac> <diam> [<dom_number>]`
Specifies that one particle domain should be randomly filled with spherical granules with specified diameter `<diam>` and volume fraction `<vol_frac>` (§8.5). Domain number to fill is given by the last optional argument.
Default `<dom_number>`: `1`

`-grid <nx> [<ny> <nz>]`
Sets dimensions of the computation grid (§8.2). Arguments should be even integers (otherwise corrected automatically by **ADDA**). In most cases `<ny>` and `<nz>` can be omitted (they are automatically determined by `<nx>` based on the proportions of the scatterer). This command line option is not relevant when particle geometry is read from a file (`-shape read`, §8.3). If `-jagged` option is used the grid dimension is effectively multiplied by the specified number (§8.3).
Default: `16` (if neither `-size` nor `-eq_rad` are specified) or defined by `-size` or `-eq_rad`, `-lambda`, and `-dpl`.

`-h [<opt> [<subopt>]]`
Shows help. If used without arguments, **ADDA** shows a list of all available command line options. If first argument is specified, help on specific command line option `<opt>` is shown (only the name of the option should be given without preceding dash). For some options (e.g. `-beam` or `-shape`) specific help on a particular suboption `<subopt>` may be shown.
Example: shape coated

`-int {poi|fcd|fcd_st|so}`
Sets prescription to calculate interaction term (§11.2). The SO formulation can not be used with `-anisotr` (§8.3). `fcd` requires dpl to be larger than 2.
Default: `poi`

`-iter {cgnr|bicg|bicgstab|qmr}`
Sets the iterative solver (§13.1).
Default: `qmr`

`-jagged <arg>`
Sets a size of a big dipole in units of small dipoles (§8.3), integer. It is used to improve the discretization of the particle without changing the shape.
Default: `1`

`-lambda <arg>`
Sets incident wavelength in μm (§8.2), float.
Default: 2π

`-m {<m1Re> <m1Im> […]|`
`<m1xxRe> <m1xxIm> <m1yyRe> <m1yyIm> <m1zzRe> <m1zzIm> […]}`
Sets refractive indices (§8.3), float. Each pair of arguments specifies real and imaginary parts of the refractive index of one of the domains. If `-anisotr` is specified, three refractive indices correspond to one domain (diagonal elements of the refractive index tensor in particle reference frame). None of the given refractive indices may be equal to 1.
Default: `1.5 0`

`-maxiter <arg>`
Sets the maximum number of iterations of the iterative solver, integer.
Default: very large, not realistic value (§13.1).

`-no_reduced_fft`
Do not use symmetry of the interaction matrix to reduce the storage space for the Fourier-transformed matrix (§13.2).

`-no_vol_cor`
Do not use "dpl (volume) correction" (§8.3). If this option is given, **ADDA** will try to match size of the dipole grid along *x*-axis to that of the particle, either given by `-size` or calculated analytically from `-eq_rad`. Otherwise (by default) **ADDA** will try to match the volumes, using either `-eq_rad` or the value calculated analytically from `-size`.

`-ntheta <arg>`
Sets the number of intervals into which range of scattering angles [0°,180°] is equally divided (§12.1), integer. This is used for scattering angles in *yz*-plane. If particle is not symmetric (§8.7) and orientation averaging (§9.2) is not used, the range is extended to 360 degrees (with the same length of elementary interval).
Default: from 90 to 720 depending on the size of the computational grid (§8.2).

`-opt {speed|mem}`
Sets whether **ADDA** should optimize itself for maximum speed or for minimum memory usage (§7).
Default: `speed`

`-orient {<alpha> <beta> <gamma>|avg [<filename>]}`
Either sets an orientation of the particle by three Euler angles α, β, γ (§9.1) or specifies that orientation averaging should be performed (§9.2). `<filename>` sets a file with parameters for orientation averaging (input format is described in §B.2).
Default orientation: `0 0 0`
Default <filename>: avg_params.dat

`-phi_integr <arg>`
Turns on and specifies the type of Mueller matrix integration over azimuthal angle φ (§12.1). `<arg>` is an integer from 1 to 31, each bit of which, from lowest to highest,

indicates whether the integration should be performed with multipliers 1, $\cos(2\varphi)$, $\sin(2\varphi)$, $\cos(4\varphi)$, and $\sin(4\varphi)$ respectively.
Examples: `1` (one integration with no multipliers), `6` (two integration with $\cos(2\varphi)$ and $\sin(2\varphi)$ multipliers).

`-pol {cm|rrc|ldr [avgpol]|cldr|fcd|so}`
Type of polarization prescription (§11.1). An optional flag `avg` can be added for LDR – it specifies that LDR polarizability should be averaged over incident polarizations. CLDR and SO can not be used with `-anisotr` (§8.3). `fcd` requires dpl to be larger than 2.
Default: `ldr` (without averaging).

`-prognose`
Deprecated command line option. Use `-prognosis` instead.

`-prognosis`
Do not actually perform simulation (not even memory allocation) but only estimate the required RAM (§7). Implies `-test`.

`-prop <x> <y> <z>`
Sets propagation direction of incident radiation (§10.1), float. Normalization (to the unity vector) is performed automatically by **ADDA**.
Default: `0 0 1`

`-save_geom [<filename>]`
Saves dipole configuration to a file `<filename>` (a path relative to the output directory, §8.3). Output format is determined by `–sg_format` and described in §C.10. This option can be used with `-prognosis`.
Default: `<type>.geom` (`<type>` is shape name).

`-scat {dr|fin|so}`
Sets prescription to calculate scattering quantities (§11.3). The SO formulation can not be used with `-anisotr` (§8.3).
Default: `dr`

`-scat_grid_inp <filename>`
Specifies a file with parameters of the grid of scattering angles for calculating Mueller matrix (possibly integrated over φ, §12.1). Input format is described in §B.4.
Default: scat_params.dat

`-sg_format {text|text_ext|ddscat}`
Specifies format for saving geometry files (§8.3). First two are **ADDA** default formats for single- and multi-domain particles respectively. DDSCAT 6.1 format corresponds to its shape option `FRMFIL` and output of `calltarget` utility (§C.10).
Default: `text`

`-shape {axisymmetric <filename>|box [<y/x> <z/x>]|capsule <h/d>|`
`coated <d`$_{in}$`/d> [<x/d> <y/d> <z/d>]|cylinder <h/d>|egg <eps> <nu>|`
`ellipsoid <y/x> <z/x>|line| rbc <h/d> <b/d> <c/d>|read <filename>|`
`sphere|spherebox <d`$_{sph}$`/D`$_x$`>}`
Sets shape of the particle, either predefined or "`read`" from file (§8.3). All the parameters of predefined shapes are floats described in detail in §8.4, except for file names.
Default: `sphere`

`-size <arg>`
: Sets the size of the computational grid along the *x*-axis in μm (§8.2), float. If default wavelength is used, this option specifies the "size parameter" of the computational grid. Default: determined by the value of `-eq_rad` or by `-grid`, `-dpl`, and `-lambda`. Size is defined by some shapes themselves, then this option can be used to override the internal specification and scale the shape.

`-store_beam`
: Save incident electric fields to a file (§10). Output format is described in §C.8

`-store_dip_pol`
: Save dipole polarizations to a file (§12.4). Output format is described in §C.8.

`-store_force`
: Calculate the radiation force on each dipole (§12.3). Requires `-Cpr_mat`.

`-store_grans`
: Save granule coordinates to a file, if `-granul` option is used (§8.5). Output format is described in §C.11.

`-store_int_field`
: Save internal fields to a file (§12.4). Output format is described in §C.8.

`-store_scat_grid`
: Calculate Mueller matrix for a grid of scattering angles and save it to a file (§12.1). Output format is described in §C.5.

`-sym {auto|no|enf}`
: Automatically determine particle symmetries (`auto`), do not take them into account (`no`) or enforce them (`enf`, §8.7).
Default: `auto`

`-test`
: Begin name of the output directory with `test` instead of `run` (§C.3)

`-V`
: Show **ADDA** version, compiler used to build this executable, and copyright information (§5.1).

`-vec`
: Calculate the not-normalized asymmetry vector (§12.2).

`-yz`
: Calculate the Mueller matrix in *yz*-plane even if it is calculated for a scattering grid (§12.1). If the latter option is not enabled, scattering in *yz*-plane is always calculated. Output format is described in §C.5.

B Input Files

All the input files should be located in the directory, in which **ADDA** is executed. Exceptions are the files specified in the command line – they may be located in a different directory. Some auxiliary files that may be required for **ADDA** execution (but which should not be modified by user) are described in §D. Comments can be used in most of the input files, they are defined as lines that start with # character. In most cases **ADDA** will detect incompatible format and terminate with an error message, consistency checks are also performed (producing error messages if necessary, §C.1). If you notice that particular input files cause abrupt termination or any erroneous behavior of **ADDA**, please communicate it to the authors. All files in this section are in ASCII format to ensure portability.

B.1 ExpCount

This is a very simple file, consisting of a single number ("run number"). In the beginning of its execution **ADDA** reads this number, increments it and saves back to the file. The read number appears in the name of the output directory (§C.3). The name of the `ExpCount` file can not be changed, however this file is not required. If it does not exist **ADDA** creates the `ExpCount` file and saves "1" in it. The purpose of the run number is two-fold: to provide convenience in sorting and analysis of output directories and guaranteeing that the name of the output directory is unique, so that **ADDA** will not overwrite any valuable data by its output. The first task (convenience) can be influenced by the user, who may change the number in `ExpCount` manually or delete this file to restart numbering.

The uniqueness of the directory name is a bit tricky, when several instances of **ADDA** run in parallel (each instance may be in sequential or parallel mode). It is possible albeit improbable that one instance of **ADDA** will read `ExpCount` between the other instance reads and updates the file. Then both instances will read the same run number. It may lead, though not necessarily, to the same name of output directories of these instances. On systems employing PBS or SGE (see §5.2) this problem is alleviated by adding a job id (which is unique) to the directory name (§C.3). Another option is used for all systems to guarantee the uniqueness of the run number – a file locking. Before reading `ExpCount` **ADDA** creates a file `ExpCount.lck` and removes it after updating `ExpCount`. All other **ADDA** instances wait until `ExpCount.lck` is removed. On POSIX systems `ExpCount.lck` is additionally locked to ensure robustness when working over the network file system, e.g. on parallel supercomputer.

Though highly improbable, permanent lock may occur under certain circumstances. That is when `ExpCount.lck` permanently exists (e.g. if **ADDA** is abruptly terminated between creating and removing this file). **ADDA** detects the permanent lock, using timeout, specified by parameters `LOCK_WAIT` (length of one wait cycles in seconds) and `MAX_LOCK_WAIT_CYCLES` (maximum number of wait cycles, after which timeout is declared), defined in the beginning of `io.c`. By default values 1 and 60 are used for this parameters respectively, i.e. if `ExpCount.lck` exists for one minute, **ADDA** exits with an error message. To solve the permanent lock problem remove `ExpCount.lck` manually.

The other potential problem of file locking is that its implementation is operating-system-dependent. **ADDA** should perform locking of `ExpCount` through lock file correctly on any POSIX-compliant or Windows operating system. However, it will produce an error during compilation if an operation system is not supported. If you experience this problem or unexplainable permanent locks occur, turn off file locking completely by uncommenting the line

```
CFLAGS += -DNOT_USE_LOCK
```

in `Makefile` and recompiling (§4). Some POSIX file systems do not support or do not allow (advanced) locking of files. **ADDA** should detect this, produce a warning, and continue without using this feature. However, the detection may take some time (up to 1 min). Therefore, if you get such warning from **ADDA** you are advised to turn off advanced file locking by uncommenting the line

```
CFLAGS += -DLOCKFILE_ONLY
```

in `Makefile` and recompiling (§4).

B.2 avg_params.dat

This file specifies parameters for orientation averaging (§9.2). It consists of three sections, each specifying the parameters for each of the Euler angles: α, β, and γ (using "*zyz*-notation"). The first section looks like

```
alpha:
min=0
max=360
Jmin=2
Jmax=4
eps=0
equiv=true
periodic=true
```

specifying minimum and maximum angles, minimum and maximum number of refinements, required accuracy, whether the minimum and maximum angles are equivalent, and whether the function is periodic on the given interval (see §13.5 to understand the meaning of these parameters). Sections for other Euler angles contain the same parameters, but start with "`beta:`" and "`gamma:`" respectively. Specified `eps` and `Jmin` are relevant for β and γ, but they are not actually used for α, because values for integration over this angle are precalculated. If `min` and `max` are the same for some angle all other parameters are ignored for it and averaging over this angle is not performed. Values of β are spaced uniformly in values of $\cos\beta$ inside the specified interval. Currently error of the integration of C_{ext} is used as a stopping criterion (it is compared to the specified `eps`).

Particle symmetries may be considered by the user to decrease the ranges of Euler angles used for averaging. For example, if particle is axisymmetric (over *z*-axis), γ is not relevant and user should set

```
gamma:
min=0
max=0
```

This will dramatically increase the speed of the orientation averaging. However even if particle is axisymmetric, its dipole representation has only 90° rotation symmetry. Hence, to be more precise, user should set `max=45;equiv=false` and decrease `Jmax` by 3. If a particle is symmetric with respect to the *xy*-plane, then the β range may be limited to [0°,90°], reducing corresponding `Jmax` by 1. Most of the particle symmetries can be employed, but that is user's responsibility to carefully account for them.

When $\beta = 0°$ or 180°, γ and α are not relevant independently but only their combination $\alpha \pm \gamma$. If α is varied in a full range [0°,360°), **ADDA** simulates only one γ value for these specific β values, saving a few evaluations of internal fields. Therefore, do *not* change the default (full) range of α, unless you have a good reason for it. It will cause all γ values to be honestly simulated. However, if you do have such a reason, please communicate it to the authors, so we can implement special optimizations for such "exotic" case.

The example of the parameter file is included in the distribution (`input/avg_params.dat`), it is commented to facilitate its editing. The order of all the

parameters is important, however comments can be inserted anywhere. A file with a different name can be used if specified in the command line (see §9.2).

B.3 alldir_params.dat

This file specifies parameters for averaging over the scattering angles for calculating integral scattering quantities (§12.2). It consists of two sections, specifying parameters for two scattering angles θ and φ. Each section is completely the same as in the `avg_params.dat` (§B.2), but starts with "`theta:`" or "`phi:`" respectively. A specified `eps` does not decrease the computational time, since all the integrated values are precalculated, but may decrease accuracy. If `min` and `max` are the same for some angle all other parameters are ignored for it and averaging over this angle is not performed. Values of θ are spaced uniformly in values of $\cos\theta$ inside the specified interval. All scattering angles are defined in the incident wave reference frame (§8.1).

Particle symmetries may be considered by the user to decrease the ranges of scattering angles used for averaging. For example, if a particle is axisymmetric (over the z-axis), range of φ can be decreased and user should set

```
phi:
min=0
max=90
equiv=false
```

and decrease `Jmax` by 2. It will significantly increase the speed of the averaging. Many of the particle symmetries can be employed, but that is user's responsibility to carefully account for them.

The example of the parameter file is included in the distribution (`input/alldir_params.dat`), it is commented to facilitate its editing. The order of all the parameters is important, however comments can be inserted anywhere. A file with a different name can be used if specified in the command line (see §12.2).

B.4 scat_params.dat

This file specifies parameters to calculate Mueller matrix on a grid of scattering angles (θ and φ), and possibly integrate result over the azimuthal angle φ (§12.1). It consists of one "global" section, two sections specifying the set of values for θ and φ, and one section for parameters of integration over φ. The first section looks like

```
global_type=grid
N=2
pairs=
0 0
30 90
…
```

First argument can be either `grid` or `pairs`. Grid is constructed as a Cartesian product of two sets of angles (described below). Pairs are specified by total number `N` and list of (θ,φ) values separated by space (each pair comes on a separate line). No comments can be inserted between "`pairs=`" and the end of the pairs list. `pairs` option is not compatible with integration over φ. The second section looks like

```
theta:
type=range
N=91
min=0
max=180
values=
0
```

```
10
…
```

`type` can be either `range` or `values`. Range is determined by `min` and `max` values, in which `N` points (including boundary points) are uniformly distributed. Values are specified by the total number `N` and a list (each value comes on a separate line). No comments can be inserted between "`values=`" and the end of the values list. A set of φ angles is defined by the similar section that starts with "`phi:`", however if integration over φ is enabled a range of φ is initialized based on the last section. This section is completely the same as in the `avg_params.dat` (§B.2), but starts with "`phi_integr:`". Specified `eps` and `Jmin` are not actually used, since all the integrated values are precalculated. All options that are not relevant for current configuration (e.g. number and list of pairs when grid of scattering angles is used) are ignored by **ADDA**, so one doesn't need to remove them from the file. All scattering angles are defined in the incident wave reference frame (§8.1).

Particle symmetries may be considered by the user to decrease e.g. the range of φ that is used for integrating. For example, if particle is symmetric with respect to the *xz*-plane, then φ range may be limited to [0°,180°], reducing corresponding `Jmax` by 1. Many of the particle symmetries can be employed, but that is user's responsibility to carefully account for them.

The example of the parameter file is included in the distribution (`input/scat_params.dat`), it is commented to facilitate its editing. The order of all the parameters is important, however comments can be inserted anywhere, except in lists of angle values or pairs. A file with a different name can be used if specified in the command line (see §12.1).

B.5 Geometry files

This file specifies the shape of the scatterer (§8.3). Two default formats are supported: for one- and multi-domain particles. A one-domain particle is described as the following:

```
#comments
0 0 1
2 1 0
…
```

First several lines are comments, after that each line contains three integers separated by space. That is *x*, *y*, and *z* coordinates of a dipole (in units of the dipole size). This coordinates can be considered relative to any (integer) reference point; **ADDA** automatically places the minimum computational box (§8.2) around all dipoles and centers it as described in §8.1. The format for multi-domain particles is similar:

```
#comments
Nmat=2
4 4 0 1
5 4 0 1
…
```

The first uncommented line specifies number of domains (different materials) and the last integer in every line specifies the domain number (1,…,`Nmat`). However, given `Nmat` is largely ignored, since its more robust estimate is the maximum of the domain numbers among all dipoles. If these two are not equal, **ADDA** produces a warning and continues with the latter, ignoring the former.

ADDA also supports DDSCAT 6.1 format corresponds to its shape option `FRMFIL` and output of `calltarget` utility [26], which looks like:

```
comments
2176 = NAT
1 0 0 = A_1 vector
0 1 0 = A_2 vector
```

```
1 1 1 = lattice spacings (d_x,d_y,d_z)/d
comments, e.g. JA  IX  IY  IZ ICOMP(x,y,z)
1 4 4 0 1 1 1
2 5 4 0 1 1 1
…
```

However, **ADDA**'s philosophy is slightly different from that of DDSCAT; in particular, shape file should describe only the dipole configuration, possibly multi-domain, but *neither* orientation nor size of the particle, which should be specified independently in the command line. Moreover, in case of anisotropic refractive index **ADDA** uses one domain number to distinguish all different refractive index tensors, even those which differ only by rotation of the tensor. Therefore, support of DDSCAT format is limited in the following respects: (a) **ADDA** completely discards first 6 lines of the shape file, i.e. it always assumes default orientation and size of the particle, unless these are specified in the command line. (b) For each "dipole" row **ADDA** uses only dipole positions (2^{nd}-4^{th} numbers) and identifier of x-component of refractive index tensor (5^{th} number) as domain number, discarding the other two components.[62]

ADDA automatically determines the format of input dipole file, analyzing its several first lines. However, with increasing number of supported formats, potential probability of autodetection failure increases. If this happens, **ADDA** produces an error because of different format of further lines. The error message contains the detected format, which can be used to diagnose the problem.

B.6 Contour file

This file specifies contour in ρz-plane defining the axisymmetric shape §8.4. It looks like:

```
#comments
0 0.5
0.2 -0.8
…
```

First several lines are comments, after that each line contains two floats defining ρ and z coordinates (in μm) of a contour point. At least three points should be specified. The contour is automatically closed by connecting the last and the first points. Linear interpolation is used between the specified points. The only requirement for the contour is that (by definition) ρ coordinates should be non-negative – **ADDA** will produce an error otherwise. For example, if a particle contains the interval of the z-axis, it is natural (although not necessary) to choose the first and the last points of the contour on the z-axis. Apart from this requirement, any set of points can be used; in particular, the contour may intersect itself or contain identical points.

[62] It is possible to analyze all three components and construct bijective mapping of all different combinations of them into domain numbers. However, in general case this mapping may be not intuitively clear, which may lead to confusion in assigning (anisotropic) refractive indices to different domains. Hence, it is currently not done. Instead **ADDA** produces a warning if it encounters anisotropic dipole in the dipole file.

C Output Files

ADDA outputs some information about its execution to `stdout` (§C.2), but most of information is saved in special files, which are created in a separate output directory (§C.3). All spreadsheet files use space as separator (both between column names and values). All files in this section are in ASCII format to ensure portability.

C.1 stderr, logerr

If any inconsistency is detected by **ADDA** during execution, it produces error and warnings. They are shown in `stderr` and duplicated in log files, as lines starting with

```
ERROR:
```

or

```
WARNING:
```

respectively. Most error messages are saved in the main log (§C.4), however in parallel mode each processor may produce specific error messages. The latter are saved in special files, named `logerr.n`, where `n` is a number of processor. In case of an error **ADDA** terminates execution, hence the errors that were detected before the output directory was created (§C.3) are not saved to file and appear only in `stderr`. Warnings indicate that probably the simulation goes not exactly the way intended by user, but **ADDA** continues execution.

C.2 stdout

ADDA's output to `stdout` is mainly designed to show the progress of the execution, when **ADDA** is run in the terminal session. More detailed information is saved to `log` (§C.4) and other output files, except for few information messages that appear only in `stdout`. The `stdout` from the sample calculation (§5.1) looks like

```
all data is saved in 'run000_sphere_g16m1_5'
0 :  0 16 16 4096 32
lambda: 6.283185307   Dipoles/lambda: 15
Required relative residual norm: 1e-005
Total number of occupied dipoles: 2176
Memory usage for MatVec matrices: 1.3 Mb
Calculating Dmatrix......
Initializing FFTW3
Total memory usage: 2.2 Mb

here we go, calc Y

CoupleConstant:0.005259037197+1.843854148e-005i
x_0 = 0
RE_000 = 1.0000000000E+000
RE_001 = 8.4752662637E-001  +
…
RE_022 = 3.1681577504E-006  +
Cext   = 135.0449046
Qext   = 3.79114961
Cabs   = -1.937494758e-016
Qabs   = -5.439177818e-018
```

It provides name of the output directory for this calculation, basic information about the scattering problem, memory requirements, progress of the iterative solver, and results for extinction and absorption cross section. It may provide more information depending on the particular simulation parameters. The second line in the example above displays some internal

information (for parallel run it shows subdivision of dipoles over the processors). When precise timing (§14.2) is enabled, all results go to `stdout`.

In some cases reading long lines in `stdout` (as well as in `stderr`) can be "inconvenient", since such lines can be broken in the middle of the word to fit the terminal width. To improve this situation **ADDA** automatic wraps lines moving whole words to the new line. It tries to determine terminal width from the environmental variable `COLUMNS`,[63] otherwise uses the value, which is defined at compilation time by the parameter `DEF_TERM_WIDTH` in the file `const.h` (80 by default).

C.3 Output directory

Output directory is generated by **ADDA** automatically to store all output files. The name of the directory has the following format

```
<type><N>_<shtype>_g<nx>m<m1Re>[id<jobid>]
```

where `<type>` is either `run` or `test`. The latter is only used if `-prognosis` (§7) is enabled or

```
-test
```

command line option is specified. `<N>` is a run number that is read from ExpCount file (§B.1) and written in a format including at least three digits. `<shtype>` is shape name (§8.3). `<nx>` is dimension of the computational grid along *x*-axis (§8.2), `<m1Re>` is real part of the first given refractive index (§8.3) written with up to 4 significant digits and decimal point replaced by "_". The last part of the name is added only if any of environmental variables `PBS_JOBID`, `JOB_ID`, or `SLURM_JOBID` is defined (they are defined by PBS, SGE, and SLURM[64] respectively, §5.2), then `<jobid>` is its value. For example, directory name may look like:

```
run000_sphere_g16m1_5
test123_box_g40m1_33id123456
```

The first one corresponds to the sample simulation (§5.1), it is included in the distribution with 3 output files in it (`sample/run000_sphere_g16m1_5`). To disable automatic naming of the output directory specify its name as an argument to the command line option

```
-dir <dirname>
```

C.4 log

This file contains most of the information that characterize the **ADDA** simulation. The `log` for the sample calculation (§5.1) is the following

```
Generated by ADDA v.0.79
The program was run on: YURKIN
command: 'd:\Maxim\Current\adda\win32\adda.exe '
lambda: 6.283185307
shape: sphere; diameter:6.734551818
box dimensions: 16x16x16
refractive index: 1.5+0i
Dipoles/lambda: 15
  (Volume correction used)
Required relative residual norm: 1e-005
Total number of occupied dipoles: 2176
Volume-equivalent size parameter: 3.367275909

---In laboratory reference frame:---
```

[63] This variable is defined on any POSIX system, but it is not necessarily exported, i.e. made available, to **ADDA**. On other systems it can be set manually if needed. It seems that using this variable is currently the most portable way.

[64] https://computing.llnl.gov/linux/slurm/

```
Incident beam: Plane wave
Incident propagation vector: (0,0,1)
Incident polarization Y(par): (0,1,0)
Incident polarization X(per): (1,0,0)

Particle orientation: default

Polarization relation: 'Lattice Dispersion Relation'
Scattering quantities formulae: 'by Draine'
Interaction term prescription: 'as Point dipoles'
FFT algorithm: FFTW3
Iterative Method: QMR (complex symmetric)
Optimization is done for maximum speed
The FFT grid is: 32x32x32
Memory usage for MatVec matrices: 1.3 Mb
Total memory usage: 2.2 Mb

here we go, calc Y

CoupleConstant:0.005259037197+1.843854148e-005i
x_0 = 0
RE_000 = 1.0000000000E+000
RE_001 = 8.4752662637E-001  +   progress = 0.152473
…
RE_022 = 3.1681577504E-006  +   progress = 0.791153

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                Timing Results
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Total number of iterations: 22
Total planes of E field calculation (each 181 points): 2

Total wall time:     0
--Everything below is processor times--
Total time:          0.3160
  Initialization time: 0.1190
    init Dmatrix         0.0140
    FFT setup:           0.1020
    make particle:       0.0000
  Internal fields:     0.1710
    one solution:        0.1710
      init solver:         0.0080
      one iteration:       0.0080
  Scattered fields:    0.0200
    one plane:           0.0100
  Other sc.quantities: 0.0010
  File I/O:            0.0050
Integration:         0.0000
```

Most of the information is self-descriptive. The hostname (on the second line) is read from the environmental variable `HOST` (in Unix) or by function `GetComputerName` (in Windows). Command line that was used to call **ADDA** is duplicated. The scatterer (§8) is completely described, then the incident beam (§10) and scatterer orientation (§9). The DDA formulation (§11) is described, then FFT algorithm (§13.2), iterative method (§13.1), and type of optimization (§7) are specified. Memory usage is given (both total and for FFT part, §7). "`calc Y`" denotes beginning of calculation for *y* incident polarization. "`CoupleConstant`"

is dipole polarizability, "x_0" denotes which initial vector is used for iterative solver (§13.1). After each iteration the relative norm of the residual is shown together with its relative decrease compared to the previous iteration (progress). A sign in between is one of +, - or -+ indicating respectively that the residual is the smallest of all the previous iterations, larger than the previous one, and smaller than the previous one but not the smallest of all the previous. log finishes with timing information (§14.1). This file may contain more information depending on the particular simulation parameters, the one that is described above is included in the distribution (sample/run000_sphere_g16m1_5/log).

C.5 mueller

This file contains results for Mueller matrix at different scattering angles (§12.1). There are a number of output files, which name starts with mueller, but they all look very similar. When scattering is computed in one scattering plane or orientation averaging (§9.2) is performed the simplest file mueller is produced:

```
    theta s11 s12 s13 s14 s21 … s44
    0.00 1.4154797793E+002 0.0000000000E+000 0.0000000000E+000 \
-5.0959034824E-012 0.0000000000E+000 … 1.4154797793E+002
    1.00 1.4140075337E+002 -5.8903788393E-003 7.3212262090E-013 \
-2.0347873246E-012 -5.8903788393E-003 … 1.4140075290E+002
    …
    180.00 2.9143742276E+000 1.0348709736E-015 -1.0104462691E-017 \
3.8122857626E-012 1.0348709736E-015 … -2.9143742276E+000
```

where "\" denotes continuation of the line. All 16 Mueller matrix elements for each scattering angle are saved. Shown is the output of the sample calculation (§5.1) – sample/run000_sphere_g16m1_5/mueller. If grid of scattering angles (any type) is calculated mueller_scatgrid is produced, which differs only by the additional column of azimuthal angle φ (and usually larger number of lines):

```
    theta phi s11 … s44
    0.00 0.00 1.4154797792E+002 … 1.4154797792E+002
    …
    180.00 360.00 2.9143742274E+000 … -2.9143742274E+000
```

This file can be produced by the command

```
    adda -store_scat_grid
```

with default scat_params.dat (§B.4). If integration over φ is enabled, up to 5 different files are produced depending on the parameters (different multipliers for integration, §12.1). They are called mueller_integr, mueller_integr_c2, mueller_integr_s2, mueller_integr_c4, and mueller_integr_s4. The format is the same as that of mueller but with addition of the column with error

```
    theta s11 … s44 RMSE(integr)
    0.00 1.4154797792E+002 … 1.4154797792E+002 2.150E-017
    …
    50.00 8.7607151503E+000 … 8.4048020875E+000 3.663E-010
    …
    180.00 2.9143742274E+000 … -2.9143742274E+000 3.601E-017
```

The shown error is relative root-mean-square error over all 16 elements of Mueller matrix,[65] integration error of each element is an estimation from the Romberg routine (§13.5). In this example the error is extremely low because of the spherical symmetry of the target and large number of integration points used (32). It is important to note that, strictly speaking,

[65] In other words, RMS absolute error is divided by RMS value.

averaging but not integration is performed over φ. The above file can be produced by the command

```
adda -phi_integr 1
```

with default `scat_params.dat` (§B.4). All scattering angles are specified with respect to the incident wave reference frame (§8.1).

C.6 CrossSec

This file contains the results for integral scattering quantities (§12.2). If orientation averaging (§9.2) is performed the result is saved to `CrossSec` file, otherwise a separate file is used for each incident polarization: `CrossSec-X` and `CrossSec-Y`. Only one file (`CrossSec-Y`) is produced if symmetry of the particle is used to simulate only one incident polarization independently (§8.7). The format is self-explanative, for example the output of the sample simulation (§5.1) looks like (`sample/run000_sphere_g16m1_5/Crosssec-Y`):

```
Cext   = 135.0449046
Qext   = 3.79114961
Cabs   = 0
Qabs   = 0
```

More results are shown in this file if additional (to default) scattering quantities are calculated.

C.7 VisFrp

This file stores the results for radiation force on each dipole (§12.3). A separate file is used for each simulated incident polarization: `VisFrp-X.dat` and `VisFrp-Y.dat`. Currently the result looks like

```
#sphere  x=3.367275909  m=1.5+0i
#number of dipoles  2176
#Forces per dipole
#r.x r.y r.z F.x F.y F.z
-0.2094395102 -1.047197551 -3.141592654 -0.001450116669
0.004378388086 -0.001487326112
…
0.2094395102 1.047197551 3.141592654 0.01038653738 \
-0.04118221215 -0.006040333419
```

However it is going to be significantly revised in the future (no comments are given here, this feature should be considered as being in development). All values are specified in the particle reference frame (§8.1). The above file can be produced by the command

```
adda -Cpr_mat -store_force
```

C.8 IntField, DipPol, and IncBeam

Internal fields, dipole polarizations (§12.4), and incident fields (§10) are saved to files `IntField-X`, `DipPol-X`, and `IncBeam-X` respectively. A separate file is used for each simulated incident polarization, their names end with different suffixes (`X` or `Y`). For instance, `IntField-Y` looks like:

```
x y z |E|^2 Ex.r Ex.i … Ez.i
-0.2094395102 -1.047197551 -3.141592654 0.6988347019 \
-0.01668015393 0.006582289815 … -0.1844808236
…
0.2094395102 1.047197551 3.141592654 5.876037053 \
0.01112798497 -0.06772761653 … -0.32940396
```

where "\" denotes continuation of the line. This file describes the dependence of the electric field vector (normalized to the incident field) on the coordinates inside the particle (in μm), suffixes `r` and `i` denote real and imaginary parts respectively. All values are specified in the

particle reference frame (§8.1). The squared norm of the electric field is presented for convenience, it is proportional to the energy density of the electric field and absorption power per unit volume for isotropic materials. The above file can be produced by the command

```
adda -store_int_field
```

Symbol used in the first row for column labels is different for different fields. It is "`E`" for the internal field, "`P`" for the dipole polarization, and "`Einc`" for the incident field. Electric fields are considered relative to the amplitude of the incident field, i.e. the values in the files are effectively dimensionless. The unit of the dipole polarizations is then the same as of volume (μm^3).

C.9 log_orient_avg and log_int

These files contain information about the 2D Romberg integration (§13.5), they are produced directly by the routine and hence have the same format. Their names are `log_orient_avg`, `log_int_Csca-X`, and `log_int_asym_x-X`, `log_int_asym_y-X`, `log_int_asym_z-X` for orientation averaging (§9.2) and calculation of scattering cross section and asymmetry vector (§12.2) respectively. The latter 4 files have different suffixes (`X` or `Y`) for different incident polarizations. For example, `log_int_Csca-Y` looks like:

```
                        PHI(rad)   cos(THETA)
    EPS                    0          0
    Refinement stages:
    Minimum                2          2
    Maximum                5          6
    lower boundary         0          -1
    upper boundary         6.28319    1
    equivalent min&max     true       false
    periodic               true       false


    Outer-Loop  Inner Loop
    Inner_qromb converged only to d=7.61895e-017 for cosine value #0
    Inner_qromb converged only to d=0 for cosine value #64
    init         64 integrand-values were used.
    Inner_qromb converged only to d=5.88e-010 for cosine value #32
    1      32 integrand-values were used.
    Inner_qromb converged only to d=4.77002e-010 for cosine value
#16
    Inner_qromb converged only to d=1.74306e-010 for cosine value
#48
    2      64 integrand-values were used.
    …
    6      1024 integrand-values were used.
    65 inner integrations did not converge.
    The outer integration did not converge
    The outer integration reached d=3.70825e-009
    In total 2080 evaluations were used
```

The first part with parameters is self-descriptive. Then for every step of outer integration (over θ), convergence of all the inner integrations and total number of integrand evaluation are shown. At the end final statistics over the whole integration is shown. An integration (outer or one of the inner) is considered converged if its estimated relative error falls below `eps` (shown in the second line), which is given in a corresponding parameter file. For orientation averaging relative error of C_{ext} is shown. For this example no adaptation is used (`eps` = 0), hence none of the integrals converge, but the reached relative errors are

informative. The range for angles (in the first column, φ or γ) is specified in *radians*. The above file can be produced by the command

```
adda -Csca
```

C.10 Geometry files

These files hold the information about the scatterer shape. They have exactly the same format as *input* geometry files (§B.5), the only difference is that **ADDA** automatically puts basic information in comments. By default, either one- or multi-domain format is used depending on the number of domains in the specified shape. For example, the command

```
adda -save_geom
```

produces the file `sphere.geom` that looks like:

```
#generated by ADDA v.0.79
#shape: 'sphere'
#box size: 16x16x16
7 5 0
…
8 10 15
```

ADDA can also save geometry in DDSCAT format, if specified by the command line option `-sg_format` (§8.3), with the following limitations: (a) particle orientation and size is always saved as default, ignoring the actual values used in a specific simulation; (b) all three identifiers of components of the refractive index tensor are saved equal to the domain number, thus effectively transforming anisotropic particles to isotropic with (possibly) larger number of domains[66] (see §B.5 for philosophical reasons of these limitations). First number saved in each row is a dipole number, starting from 1. However, in parallel mode it is a local number of a dipole for a specific processor, to which this dipole belongs (§0). Therefore, we recommend to ignore these numbers during processing of geometry files in DDSCAT format, as postulated in its manual [26].

C.11 granules

This file stores the coordinates of the granules produced by granule generator (§8.5). Currently the result looks like:

```
#generated by ADDA v.0.79
#granule diameter = 0.9980867612
-0.6693706848 1.833970089 -0.9894352851
…
-2.217014497 0.5198635377 -2.576990365
```

where each line specifies x, y, and z coordinates of a granule center in particle reference frame (§8.1). However the format should be considered in development and it may change in the future, e.g., to correspond the input shape format for cluster of spheres (when the latter will be developed). The above file, with different numbers due to random placement, can be produced by the command

```
adda -granul 0.1 1 -store_grans -prognosis
```

[66] It is possible to remove this limitation, e.g. by mapping each domain number n into trio $3n-2$, $3n-1$, $3n$. However, it will cause a lot of unnecessary complications for isotropic particles.

D Auxiliary Files

These files can be used by **ADDA** under certain circumstance, however they are not intended to be inspected or modified by user (except `chp.log`, §D.2). This is just general information to facilitate the understanding of how the **ADDA** actually works.

D.1 tables/

This is the directory that contains precalculated tables of some integrals, that are used by **ADDA** for SO prescription for the interaction term (§11.2). Since the latter feature is still in development, the tables and their format may change in future. Currently it contains 10 files: `t1f.dat`, … , `t10f.dat`. These files are in text format to be completely portable, and occupy totally about 150 kB of disk space. They are included in the distribution (`input/tables/`).

D.2 Checkpoint files

These files are produced when **ADDA** saves a checkpoint (§13.4), and are used when **ADDA** loads one. By default the directory `chpoint` is used to store all files, however different directory name can be specified in a command line (§13.4). This directory contains one text file `chp.log`, which contains some information for the user. Currently it is only the name of the directory, where the output of the **ADDA** instance, which produced the checkpoint, was saved. Each processor (number *k*) produces a file named `chp.`*k* that contains the information, completely describing the state of the iterative solver on this processor, in binary format.[67]

[67] Hence it is not necessarily portable between different systems.

E Comparison with Other Codes

E.1 Comparison with other DDA codes

Comparison of four different DDA codes, including **ADDA** 0.7a and DDSCAT 6.1, was performed by Penttila *et al.* [43] for simulation of different wavelength-sized particles both in single orientation and averaged over all orientations. Here we present a brief summary and a few sample results from this paper. **ADDA** is faster than other codes by a factor of 2.3 to 16 and it is the least memory consuming code. Results of simulations of light scattering by a sphere in a fixed orientation are shown in Fig. 7. All scatterers considered in this section has volume-equivalent size parameter 5.1 and refractive index 1.313. SIRRI and ZDD denote codes by Simulintu Oy in Finland and Zubko and Shkuratov respectively. All codes result in almost identical relative error of $S_{11}(\theta)$, as compared with the Mie theory, showing that the codes use the same formulation of DDA, except SIRRI that is slightly different.

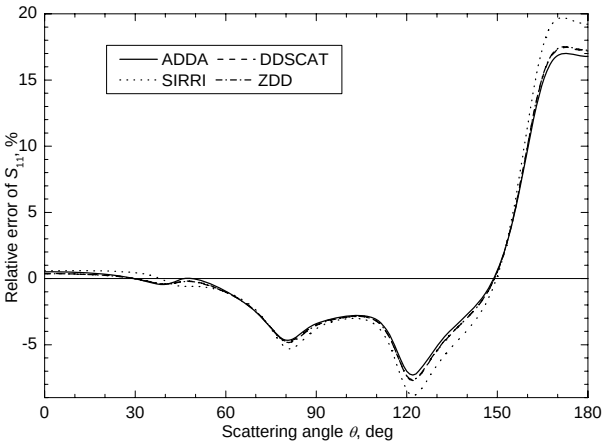

Fig. 7. Comparison of relative errors of $S_{11}(\theta)$ for a sphere, simulated using four different DDA code. Adapted from [43], details are given in the text.

On contrary, orientation averaging schemes of the codes are different. Orientation averaged results for a randomly constructed aggregate of 50 spheres are shown in Fig. 8 (here and below reference results are obtained using the T-matrix method). One can see that results differ, and the errors of **ADDA** results are the largest. That is explained by the fact that for **ADDA** we used 16×9×8 sample points for Euler angles α, β, and γ respectively (§9.2). Other codes used less α sample angles (11, 4, and 1) but more β and γ sample angles, resulting in approximately the same total number of sample angles. Therefore, **ADDA** is faster at expense of possible deterioration of accuracy. However, for other asymmetric particles studied in [43] **ADDA** results in comparable to other codes accuracy even with current settings (particle showed here is the least favorable for **ADDA**).

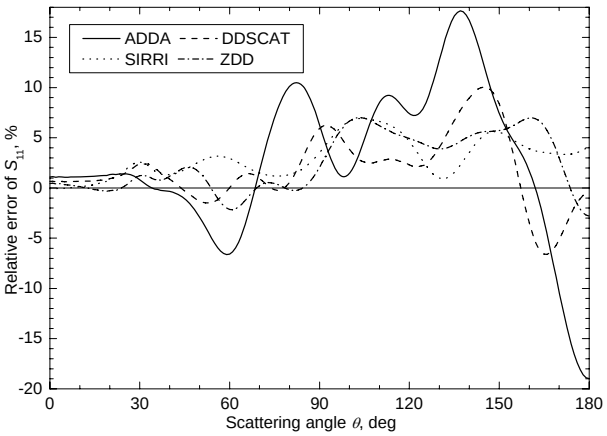

Fig. 8. Same as Fig. 7 but for a 50-sphere aggregate. Adapted from [43].

The results for axisymmetric particles are principally different. Here we consider only a prolate spheroid with aspect ratio 2 (Fig. 9). One can see that **ADDA** result (the first curve) is much less accurate than other codes. The accuracy of **ADDA** is unusually bad, even compared to Fig. 8. In the following we explain this fact. For axisymmetric particles the dipole array is

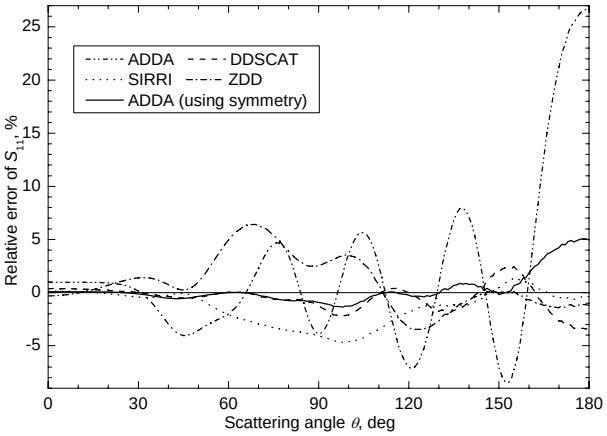

Fig. 9. Same as Fig. 7 but for a spheroid. Adapted from [43] with addition of new **ADDA** results.

symmetric with respect to 90° rotation over the symmetry axis. Moreover, the range from 0° to 90° for γ is mirror symmetric with respect to the 45°. Hence, the most efficient way of orientation averaging for such particles is to sample γ uniformly in the range from 0° to 45°, and also to use less γ sample angles but more β sample angles. Sampling by other codes satisfies these conditions. The situation for **ADDA** is remarkably different. 8 γ angles are used, which are spaced with interval 45°, hence they map in only two distinct angles (0° and 45°) inside the reduced range. That means that for axisymmetric particles **ADDA** repeats calculation for four equivalent sets of orientations. Moreover, Romberg integration, which is a higher order method, performs unsatisfactory when applied to a range that contains several periods of the periodic function to be integrated and only a few points per each period. An analogous discussion applies to the β angle since both spheroid and cylinder have a symmetry plane perpendicular to the symmetry axis.

It is important to note that **ADDA** is perfectly suitable to calculate orientation averaging of axisymmetric particles efficiently, but it requires a user to manually modify the sampling intervals for Euler angles in the input files. For instance, for the discussed scatterers the choice of 17 β angles from 0° to 90° and 5 γ angles from 0° to 45° is appropriate. The result of this tuned orientation averaging for spheroid, obtained using **ADDA** 0.76, is shown as the last curve in Fig. 9. It requires about 20% more computational time than the initial scheme, but still it is significantly faster than other codes. One can see that its accuracy is comparable to or even better than that of other codes. This result was not included in [43], because such modifications of the input files were considered a 'fine-tuning'. DDSCAT may also benefit from reduction of angle ranges, but in much lesser extent (in these particular cases).

Therefore, we may conclude that **ADDA** is clearly superior for particles in single orientation. Its orientation averaging scheme is not that flexible, especially when small number of sample angles is used. However, in most cases it results in accuracy not worse than that of other codes, if particle symmetries are properly accounted for. Moreover, inflexibility is compensated by the adaptability of the **ADDA** orientation averaging scheme and its ability to estimate the averaging error (§9.2).

E.2 Simulation of a Gaussian beam

To validate the part of **ADDA**, which simulates scattering by a Gaussian beam (§10.2), we conducted a simple test and compared results with Multiple Multipole Program (MMP). The MMP results were provided by Roman Schuh and Thomas Wriedt. Sphere with size parameter 5 and refractive index 1.5 is illuminated with a Gaussian beam with wavelength 1 μm, width 2 μm, and position of the beam center (1,1,1) μm. The beam propagates along the z-axis. We calculated $S_{11}(\theta)$ in yz-plane, scattering angle θ was considered from z-axis to y-axis. We used 64 dipoles per diameter of the sphere. **ADDA** 0.76 was used with a command line:

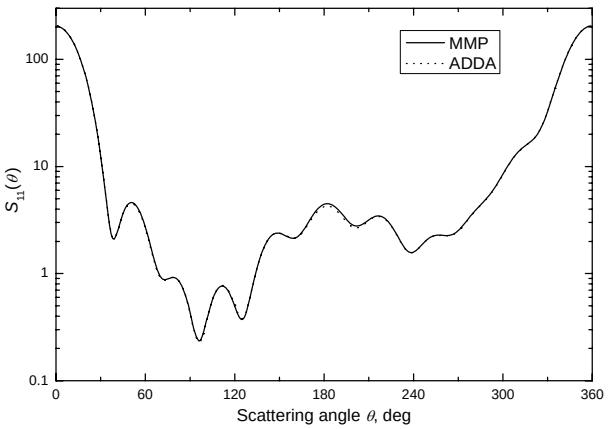

Fig. 10. Comparison of **ADDA** (dotted line) with MMP (solid line) for a sphere illuminated by a Gaussian beams. Parameters are given in the text. Dependence of $S_{11}(\theta)$ in logarithmic scale is shown.

```
adda -grid 64 -lambda 1 -size 1.59155 -beam barton5 2 1 1 1
-ntheta 180
```

The results are shown in Fig. 10, showing perfect agreement between two methods.

E.3 Comparison between the DDA and the FDTD

A systematic comparison of the DDA and the finite difference time domain (FDTD) method was performed for large dielectric scatterers by Yurkin *et al.* [44]. Here we summarize these results. Simulations were performed for spheres with x from 20 to 80 and m from 1.02 to 2 (all refractive indices had a fixed negligible imaginary part). Since both methods are potentially exact if infinite discretization is employed, a certain accuracy was required – relative error of Q_{ext} less than 1% and the root mean square relative error of S_{11} over the whole range of θ less than 25%. Then simulation times were compared. **ADDA** v.0.76 was used as DDA implementation, and FDTD implementation was developed in the Biomedical Laser Laboratory at East Carolina University, based on the algorithms described by Yang and Liou with numerical dispersion correction (see [44] for details).

The DDA is more than an order of magnitude faster for $m \le 1.2$ and $x > 30$, while for $m \ge 1.7$ the FDTD is faster by the same extent. $m = 1.4$ is a boundary value, for which both methods perform comparably. The DDA errors of $S_{11}(\theta)$ for $m = 1.02$ are mostly due to the shape errors, which are expected to be smaller for rough and/or inhomogeneous particles. Simulations for a few sample biological cells lead to the same conclusions. Although these conclusions depend on particular scattering quantity and on the implementations of both methods, they will not change principally unless a major improvement of one of the method is made. However, for general complex refractive indices the results of the comparison are expected to be significantly different.

F How to Modify the Source Code

Minor issues, how the code can be tuned for a particular problem/hardware, such as changing of values of global constants, are discussed in the main part of the manual. Here some advanced issues are discussed.

F.1 Adding a new predefined shape

See §8.4 for a description of implemented predefined shapes. All places in the code, where something should be added, are commented. These comments start with "`TO ADD NEW SHAPE`" and contain detailed description; the text below is less detailed. Added code should be analogous to shapes already present. The procedure is the following:

- Add new shape descriptor in the file `const.h`. Starting descriptor name with `SH_` is recommended, as well as adding a descriptive comment.
- Add information about the new shape in the file `param.c`. It should be a new row in the constant array `shape_opt`. This line should contain: the shape name (it will be used in a command line), usage string (the description of possible input parameters), help string (it will be shown when `-h` option is used, §5.1), possible number of parameters (real values), shape descriptor. If this shape can accept variable number of parameters use `UNDEF` instead of a number and then check the number of parameters explicitly in function `PARSE_FUNC(shape)` (below in the same file). If shape accepts a single parameter with a file name, use `FNAME_ARG` instead.
- Add initialization of the new shape in the end of "`else if`" sequence in the function `InitShape` in the file `make_particle.c`. This initialization should save all parameters from the array `sh_pars` to local variables, which should be defined in the beginning of the file (the place for that is shown by a comment). Then test all input parameters for consistency (for that you are encouraged to use functions from `param.h` since they would automatically produce informative output in case of error). If the shape breaks any symmetry (§8.7), corresponding variables should be set to `false`. Currently only two symmetries are relevant, however it is recommended to consider all of them (see the comments in the code) for future compatibility. Then initialize the following variables: 1) `sh_form_str` – descriptive string that would be shown in the `log`; 2) either `yx_ratio` (D_y/D_x, §8.2) or `n_boxY` (n_y); 3) either `zx_ratio` (D_z/D_x) or `n_boxZ` (n_z); 3) `Nmat_need` – number of different domains in this shape (§8.4); 4) `volume_ratio` – ratio of particle volume (in units of dipoles) to n_x^3, it is used for dpl correction (§8.3); 5) `n_sizeX` – absolute size of the particle, defined by shape (only when relevant); 6) all other auxiliary variables that are used in shape generation (below). If you do not initialize `yx_ratio` and/or `zx_ratio` set it explicitly to `UNDEF`. If you need temporary local variables (which are used only in this part of the code), either use `tmp1` – `tmp3` or define your own (with more informative names) in the beginning of `InitShape` function. Also (rarely) if the shape defines dimension of the computational grid or absolute size of the particle, correct values of `box_det_sh` and `size_det_sh` in the same location.
- Add definition of the new shape in the end of "`else if`" sequence in the function `MakeParticle` in the file `make_particle.c`. This definition should set the variable `mat` – index of domain – for each point, specified by (`xr`,`yr`,`zr`) – coordinates divided by D_x (§8.2). Do not set `mat` for void dipoles. If you need temporary local variables (which are used only in this part of the code), either use

`tmp1` – `tmp3` or define your own (with more informative names) in the beginning of `MakeParticle` function.

If you add a new predefined shape to **ADDA** according to the described procedure, please write to the discussion group,[5] so your code can be incorporated in future releases.

F.2 Adding a new predefined beam type

See §10.2 for a description of implemented beam types. All places in the code, where something should be added, are commented. These comments start with "`TO ADD NEW BEAM`" and contain detailed description; the text below is less detailed. Added code should be analogous to beam types already present. The procedure is the following:

- Add new beam descriptor in the file `const.h`. Starting descriptor name with `B_` is recommended, as well as adding a descriptive comment.
- Add information about the new beam type in the file `param.c`. It should be a new row in the constant array `beam_opt`. This line should contain: the beam name (it will be used in a command line), usage string (the description of possible input parameters), help string (it will be shown when `-h` option is used, §5.1), possible number of parameters (real values), beam descriptor. If this beam type can accept variable number of parameters use `UNDEF` instead of a number and then check the number of parameters explicitly in function `PARSE_FUNC(beam)` (below in the same file). If beam type accepts a single parameter with a file name, use `FNAME_ARG` instead.
- Add initialization of the new beam type in the end of "`else if`" sequence in the function `InitBeam` in the file `GenerateB.c`. This initialization should save all parameters from the array `beam_pars` to local variables, which should be defined in the beginning of the file (the place for that is shown by a comment). Then test all input parameters for consistency (for that you are encouraged to use functions from `param.h` since they would automatically produce informative output in case of error). If the shape breaks any symmetry (§8.7), corresponding variables should be set to `false`. Currently only two symmetries are relevant, however it is recommended to consider all of them (see the comments in the code) for future compatibility. Then initialize the following variables: 1) `beam_descr` – descriptive string that would be shown in the `log`; 2) `beam_asym` – whether beam center does not coincide with the reference frame origin. If it is set to `true`, then set also `beam_center_0` – 3D radius-vector of beam center in the laboratory reference frame (§8.1); 3) all other auxiliary variables that are used in beam generation (below). If you need temporary local variables (which are used only in this part of the code) define them in the beginning of `InitBeam` function.
- Add definition of the new beam type in the end of "`else if`" sequence in the function `GenerateB` in the file `GenerateB.c`. This definition should set complex vector `b`, describing the incident field in the particle reference frame. It is set inside the cycle for each dipole of the particle and is calculated using 1) `DipoleCoord` – array of dipole coordinates; 2) `prop` – propagation direction of the incident field; 3) `ex` – direction of incident polarization; 4) `ey` – complementary unity vector of polarization (orthogonal to both `prop` and `ex`); 5) `beam_center` – beam center in the particle reference frame (automatically calculated from `beam_center_0` defined in `InitBeam`). If you need temporary local variables (which are used only in this part of the code), either use `t1` – `t8` or define your own (with more informative names) in the beginning of `GenerateB` function.

- Take a look at function `ExtCross` in file `crosssec.c`. Used formulae for non-plane beams assume that the amplitude of the beam is unity at the focal point. So either make sure that the new beam type satisfies this condition or add other formulae to this function.

If you add a new predefined beam type to **ADDA** according to the described procedure, please write to the discussion group,[5] so your code can be incorporated in future releases.

F.3 Adding a new command line option

The complexity of implementing a new command line option strongly depends on the required functionality. The logic may be described by as small as a few lines of code or may require rewriting large parts of **ADDA**. Here we describe a common part of any new command line option, which integrates it into command line parser, consistency tester and internal help system (§5.1). Corresponding changes in the code should be made in several places in `param.c`. These places are commented starting with "`TO ADD NEW COMMAND LINE OPTION`" and contain detailed description; the text below is less detailed. Added code should be analogous to command line parameters already present. The procedure is the following:

- Choose name for a new command line option: `option_name`.
- Add a function prototype `PARSE_FUNC(option_name);`[68] to the existing list of prototypes.
- Add a row to definition of static structure `options`, containing `PAR(option_name)`, usage string (the description of possible input parameters), help string (it will be shown when `-h` option is used, §5.1), number of arguments, pointer to suboption (if exist, `NULL` otherwise). If this command line option can accept variable number of parameters use `UNDEF` instead of a number and then check the number of parameters explicitly in function `PARSE_FUNC(option_name)` (see below). A separate suboption structure is recommended only for large options such as `-beam` and `-shape`, and it should be defined as type `static const struct subopt_struct` in the beginning of `param.c`.
- Add a function definition starting with `PARSE_FUNC(option_name)` to the existing list of definitions. This function actually implementation implements the necessary logic for the command line option or at least registers that it was met during parsing of the command line. Typically, this function should check number of parameters (if `UNDEF` was used in the option definition, see above), parse parameters and check them for consistency (if applicable), and set the values of some internal variables or flags. If a new command line option contains suboptions, you should also parse them using the suboption structure and check for consistency. For working with input parameters you are encouraged to use functions from `param.h` and `param.c` since they are designed to be overflow-safe and would automatically produce informative output in case of error.
- New variables required for implementation of the command line option should be defined in the beginning of `param.c` and initialized in function `InitVariables`. However, you may also define variables as semi-global or global if they are used in two or more than two source files respectively.
- If a new command line option may potentially conflict or interact with other options, address this issue in function `VariablesInterconnect`.

[68] `PARSE_FUNC(…)` and `PAR(…)` are macros, which are defined in `param.c` and used for conciseness.

- If any output in `log` or `stdout` should be produced based on the presence command line option or on the values of its parameters, it should be implemented in function `PrintInfo`.

If you add a new command line option to **ADDA** according to the described procedure, please write to the discussion group.[5] Probably this new option will be useful for other users, so incorporating your code in future releases will be beneficial for the community.

G Tutorial

This tutorial is a slightly modified version of the one, carried out for participants of the DDA workshop in Bremen at 23.03.2007. It is based on the **ADDA** v.0.76, but most probably applies to all later versions. References to relative sections of the manual are included throughout the tutorial, but please use them only if you are really stuck. The tutorial is designed to be self-contained and we believe it is best enjoyed this way. It is assumed that you have compiled the sequential version of **ADDA** or have downloaded executables (§4). Before starting please make sure that you have the following files in your working directory:

`adda` or `adda.exe`
`alldir_params.dat`
`avg_params.dat`
`libfftw3-3.dll` (for Windows)

In the command lines of this tutorial it is assumed that **ADDA** can be run by typing `adda`, when positioned in the working directory. This can differ depending on the particular operating system (§5.1).

1. Running **ADDA** is simple. Go and try it:
   ```
   adda
   ```

This simulates light scattering by a default sphere (size parameter $x = 3.367$, refractive index $m = 1.5$, size of computational box along x-axis $n_x = 16$ dipoles). You see the basic output of **ADDA** on the screen (§C.2). It shows the directory name used for all output and problem parameters: incident wavelength λ, number of dipoles per λ 'dpl', required relative residual norm of the iterative solver ε. Then some derived values are shown, such as total number of non-void dipoles in the computational box and total memory requirements. It is followed by the progress report of the iterative solver, which shows the relative residual norm for each iteration. Finally, the extinction and absorption cross sections (C_{ext}, C_{abs}) and efficiencies (Q_{ext}, Q_{abs}) are shown.

This output is not designed to be complete but rather to provide the information on **ADDA** progress when simulation takes longer than a few seconds. A systematic overview of the simulation is provided in the output directory. It is located in the working directory and is named like `run000_sphere_g16m1_5` (§C.3). Enter into this directory and observe three files: `log`, `mueller`, `CrossSec-Y`. The file `log` (§C.4) contains a detailed description of problem parameters, the convergence behavior of the iterative solver, used memory, and computational time divided into several different parts. The file `mueller` (§C.5) contains the calculated Mueller matrix as a function of polar angle in one scattering plane. The file `CrossSec-Y` (§C.6) contains the calculated integral scattering quantities, e.g. cross sections, for incident polarization along y-axis.

2. Most of the problem parameters can be specified in the command line. Try the following:
   ```
   adda -grid 16 -m 1.5 0 -dpl 15
   ```

The result is essentially the same as before, because the specified parameters (n_x, m, and dpl) are set to the same as default values.

3. To simulate the scattering of a sphere with diameter equal to λ type in
   ```
   adda -grid 16 -m 1.5 0 -size 1 -lambda 1
   ```

This sets both D_x (size of the computational box along x-axis) and λ equal to 1 μm. However, the relevant value is the D_x/λ or some kind of size parameter (§8.2).

4. If λ is not given in the command line, the default value 2π μm is used. That means that `-size` effectively specifies the value of kD_x (k is free space wavenumber). The most natural way to run **ADDA** is to specify only the size parameter and the refractive index of a scatterer:

```
adda -m 1.5 0 -size 6
```

This simulates light scattering by a sphere with $m = 1.5$ and size parameter 3. The discretization of the scatterer is managed automatically by **ADDA**, for that "rule of thumb" is used: dpl = 10|m|.

5. The performance of **ADDA** significantly depends on the refractive index (§8.3). Try the index-matching scatterer:

```
adda -m 1.05 0.1 -size 6
```

The convergence is now much faster (5 instead of 18 iterations). Since you specified nonzero imaginary part of refractive index, nonzero C_{abs} is obtained.

6. Now it is time to ask **ADDA** some questions. Type in

```
adda -h
```

As expected, you get help, more specifically a list of all possible command line options. **ADDA** also informs you, how you can get more detailed help on any option .

7. You are probably tired of spheres, since that is not what you want to use DDA for. To see, which other shapes **ADDA** can handle, run:

```
adda -h shape
```

The list is quite short (§8.3); however, it contains an option `read`, which allows specifying almost any shape you like in a file.

8. Get information on 'coated sphere' predefined shape:

```
adda -h shape coated
```

9. Now you can simulate scattering by sphere with a spherical inclusion, not necessarily concentric. Run

```
adda -size 6 -shape coated 0.5 0.2 0 0
```

This should simulate a sphere with $x = 3$, having spherical inclusion with twice less x. Inner sphere is also shifted by 20% of outer sphere diameter along the x-axis. However, the above command line results in an error! **ADDA** detects almost all possible errors in input parameters and produces informative error messages (§C.1). Read carefully the error message.

10. **ADDA** uses default $m = 1.5$ if it is not specified, but only for one-domain scatterers. Hence, you must specify m for both core and outer spheres:

```
adda -size 6 -shape coated 0.5 0.2 0 0 -m 1.05 0 1.2 0
```

Now the simulation goes on fluently, but you see the output of *two* instances of the iterative solver. That is because the scatterer is not spherically symmetric, for the first time in the tutorial. In this general case scattering of two incident polarizations should be simulated to get the Mueller matrix. The good thing about **ADDA** is that it cuts the simulation time twice for scatterers that are symmetric with regard to rotation by 90° over the z-axis (§8.7).

Look inside the last output directory. There you will find an additional file `CrossSec-X`, which contains integral scattering quantities for incident x-polarization.

11. Usually, C_{sca} is calculated as $C_{ext} - C_{abs}$. However, in some cases it is desirable to calculate it directly by integrating the amplitude of the scattered field over the solid angle:

```
adda -Csca
```

The parameters of integration are specified in the file `alldir_params.dat` (§B.3), but it is not considered in the tutorial – the default parameters are used. **ADDA** uses Romberg integration (§13.5) that provides the estimation of the integration error, which is shown in the output. This error is extremely small because of the spherical symmetry of the scatterer. However, if you look at the values of C_{ext} and C_{sca}, which should be equal for non-absorbing particles, relative difference between them is few orders of magnitude larger than this error. That is because the iterative solver itself introduces an error (§13.1), which is about 10^{-5} by default.

12. Decrease the required ε to 10^{-10}:

```
adda -Csca -eps 10
```

The simulation time becomes about 1.5 times longer and results in C_{ext} and C_{sca} being identical up to all shown digits.

13. Now switch to the orientation averaging (§9.2). To make it non-trivial consider a general ellipsoid. First read the information on the predefined shape 'ellipsoid':

```
adda -h shape ellipsoid
```

14. Then simulate light scattering by an ellipsoid averaged over all possible orientations. Consider a small one to make it fast (computational grid 4×8×12 dipoles, dpl = 15):

```
adda -grid 4 -shape ellipsoid 2 3 -orient avg
```

The simulation takes some time producing a lot of output (it shows the convergence progress of each instance of the iterative solver). Consider the final part of this output. It shows the error estimate of averaging over the α Euler angle, which is done in a single DDA run. Then the status of the convergence for two other Euler angles is shown. They both converged to the relative error of 10^{-3} (measured for Q_{ext}). This value together with other parameters for orientation averaging is specified in the file `avg_params.dat` (§B.2), which is also not considered in the tutorial.

The powerful part of orientation averaging in **ADDA** is its adaptability. It reaches the specified accuracy using the minimum number of single DDA simulations (this number is shown in the output). More detailed information on the orientational averaging can be found in the file `log_orient_avg` in the output folder (§C.9).

15. One of the "jewels" in **ADDA** is the granule generator (§8.5). You may never use it in the future, but it is extremely powerful for certain applications. Suppose you want to simulate light scattering by a sphere randomly filled with small spherical granules. You need only to specify diameters of the sphere and the granules, volume fraction of the granules, and two refractive indices:

```
adda -size 16 -granul 0.3 1 -m 1.05 0 1.2 0
```

This corresponds to granules with size parameter 0.5, volume fraction 0.3, and $m = 1.2$ in sphere with $x = 8$ and $m = 1.05$. You can find some information about the progress of random granule placement in the output.

This tutorial covers only the basics of using **ADDA** and a few selected advanced features. At the same time it does not consider a lot of other features, for example: parallel execution enabling handling very large particles (§5.2), Gaussian incident beam (§10.2), different iterative solvers (§13.1) and prescriptions for polarizability (§11), using symmetry of the scatterer to accelerate orientation averaging (§B.2), checkpoint system (§13.4), and calculation of scattering for an arbitrary set of scattering angles (§12.1). If you are not yet fed up with **ADDA**, here is a final task:

16. Calculate Q_{ext} for a spherical cell with diameter 2 μm, placed in the origin. Inside the cell there is a spherical nucleus with diameter 1 μm, centered in (0,0,0.2) μm. The outer shell (cytoplasm) is filled with granules of diameter 0.2 μm and volume fraction 0.2. The refractive indices of the cytoplasm, the nucleus, and the granules are 1.02, 1.08, and 1.2 respectively. The incident wavelength is 0.4936 μm. There is no command line provided here, however, you are encouraged to use **ADDA** help system or the manual. If you do it correctly, you should get Q_{ext} in the range 1.146 ± 0.029 (95% confidence interval). Good luck!

H A brief history of long ADDA development

The development of **ADDA** started in 1990 when the original author, Alfons Hoekstra, started with his PhD research. His assignment was to implement the coupled dipole method (as we called it back then) on a recently acquired computer at the University of Amsterdam. This was a distributed memory parallel computer. The research question was two-fold: 1) demonstrate parallel computing performance of a production code on a parallel computer and 2) use it for investigation of elastic light scattering from biological particles.

So, **ADDA** was a parallel program from the start. The computer in question was a Meiko CS1, containing a whopping 64 T800 Transputers.[69] At that time this was a revolutionary machine based on a revolutionary processor. However, there was a price to be paid. When the first version of **ADDA** was implemented, there were no compilers available on the Meiko CS1 for the standard programming languages such as C or Fortran. A bit later they arrived, but we decided not to wait and have a first implementation in a language that was co-developed with the Transputer, and this language was Occam [45]. Moreover, the Meiko was hardly equipped with modern communication libraries (like we now have in MPI), so all the routing needed to be hard coded on the machine. Below are some short code snippets to get a feeling how the first **ADDA** code looked like. Unfortunately, the Occam source codes are no longer really accessible, as the Occam editor/compiler used the concept of a folding editor (a bit like Mathematica notebooks), and this editor/compiler does not run on current versions of Unix. Below is some code to create a ring of Transputers:

```
    VAL m IS (nTransputers-1):
    CHAN OF ANY toHost, fromHost:
    [nTransputers+1] CHAN OF ringChannel linkClockWise,
linkNotClockWise :
    -- linkClockWise[0] is the link, talking clockwise between the
transputers 0-1
    -- linkNotClockWise[3] is the link between the transputers 3-4,
talking in
    -- not ClockWise direction
    PLACED PAR PROCESSOR 0 T8
    PLACE toHost AT toL2:
    PLACE fromHost AT fromL2:
    PLACE linkClockWise[0] AT toL0:
    PLACE linkNotClockWise[0] AT fromL0:
    …
```

Next, some Occam compute code:

```
    --     This procedure calculates the operator F (see     --;
    --     internal report "The Radiating Dipole", by        --;
    --     A.G. Hoekstra) which couples the dipole moment    --;
    --     located at rdip with the electric field observed  --;
    --     at robs. The result is stored in f.r and f.i,     --;
    --     the real and imaginary part of the complex        --;
    --     interaction tensor.                               --;
    --                                                       --;
    ---------------------------------------------------------;
    --                                                       --;
    --                          A.G. Hoekstra, spring 1991   --;
    --                                                       --;

      #USE maths64.lib>
      #USE cmplx.lib
```

[69] http://www.wikipedia.org/wiki/Transputer

```
      -- calls to the complex lib could be omitted
      -- this will improve the performance
      [3]REAL64 r:  -- vector from dipole to observer2
      REAL64 rabs, a.r, a.i, b.r, b.i, eikr.r, eikr.i:
      SEQ
        r[0],r[1],r[2] := (robs[0]-rdip[0]),(robs[1]-rdip[1]),
(robs[2]-rdip[2]):
        rabs := DSQRT(((r[0]*r[0]) + (r[1]*r[1])) + (r[2]*r[2])):
        r[0], r[1], r[2] := r[0]/rabs, r[1]/rabs, r[2]/rabs
    …
```

Note the use of the `SEQ` call. In Occam we were forced to specify if a computation should be done sequentially (`SEQ`) or in parallel (`PAR`). Anyway, a beautiful language, and we got good parallel performance (see e.g. [46,47]), but the code was off course absolutely not portable. We proved that the coupled dipole method could reach very high efficiencies on a parallel computer, but then we needed a more portable and maintainable code.

Around 1992 a next parallel computer arrived at the University of Amsterdam, the Parsytec CGel, equipped with 512 T805 Transputers. It had C-compilers, and a decent library for Single Program Multiple Data (SPMD) type parallel computing (the Parix library). So, end of 1992 **ADDA** was ported to C and Parix and was running on 512 transputers (see e.g. [48]). This was the time that the parallel computing community was working hard on developing SPMD libraries, and a number of competing environments were available. We used **ADDA** to compare Parix, Express and PVM[70] [49,50]. The PVM emerged as the de-facto standard, and as **ADDA** was ported to PVM, we kept that line of development and ported the code to some other parallel computers, as well as to clusters of workstations (for which PVM was originally developed).

The PVM-**ADDA** code of around 1994 was very powerful in terms of computational speed and parallel performance [14]. However, it missed one very important feature to make it a true production code. It did not include the FFT to do the matrix-vector products in the iterative solver. So, albeit parallel, the code had an unacceptable $O(N^2)$ computational complexity. The point was, at that time there were not highly portable, tailored parallel FFT's available. Fortunately, a gifted student, Michiel Grimminck, developed a parallel FFT for **ADDA**, reaching impressive performances at that time [15]. **ADDA** was also ported to MPI which is up until now, the standard for SPMD style parallel programming.

That version of **ADDA** was used for a long time. We used it to report DDA simulation of a sphere with size parameter 40 in the year 1997 [51]. This was hardly believed by the audience, as it was commonly believed that size parameters much larger than say 15 or 20 were not possible. Indeed, on single processor workstations that was more or less the limit, but on a parallel supercomputer we could move to much larger particles.

ADDA was used for many different light scattering studies, but its implementation was hardly changed. The number of pre-defined particles were slowly increased, new observables were added (e.g. the option to compute radiative forces [16]) and the code was constantly ported to new parallel machines. The later was an easy job, as the combination of a C code with MPI was easily portable.

With the arrival of Maxim Yurkin in Amsterdam in 2004 a next wave of **ADDA** development started. The code was completely redone, an optimized parallel FFT package (the FFTW) was included, a Windows version was created, finally reaching its current state. **ADDA** was then put into the public domain, and can now be used by the community at large. The history of the latter stage is documented in `doc/history` and in **ADDA** Subversion repository (§3).

[70] http://www.csm.ornl.gov/pvm/pvm_home.html