This document can be cited as: Choliy, V. 2013,
"DDscat.C++ 7.3.0 User and programmer guide"
http://arxiv.org/abs/xxxx.xxxx

DDscat.C++ 7.3.0
User and programmer guide

Vasyl Ya.Choliy
Taras Shevchenko National University of Kyiv
`Choliy.Vasyl@gmail.com`

Last revised 24 June 2013

Abstract

DDscat.C++ 7.3.0 is a freely available open-source C++ software package applying the "discrete dipole approximation" (DDA) to calculate scattering and absorption of electromagnetic waves by targets with arbitrary geometries and a complex refractive index. **DDscat.C++** is a clone of well known **DDSCAT** Fortran-90 software. We refer to **DDSCAT** as to the parent code in this document. Versions 7.3.0 of both codes have the identical functionality but the quite different implementation. Started as a teaching project, the **DDscat.C++** code differs from the parent code **DDSCAT** in programming techniques and features, essential for C++ but quite seldom in Fortran.

As **DDscat.C++** in its current version is just a clone, usage of **DDscat.C++** for electromagnetic calculations is the same as of **DDSCAT** . Please, refer to "User Guide for the Discrete Dipole Approximation Code DDSCAT 7.3" [1] to start using the code(s).

This document consists of two parts. In the first part we present Quick start guide for users who want to begin to use the code. Only differencies between **DDscat.C++** and **DDSCAT** are explained. That is why a lot of references to [1] are in the first part. The second part of the document explains programming tips for the persons who want to change the code, to add the functionality or help the author with code refactoring and debugging.

The author is grateful to thanks B.Draine and P.Flatau for positive and warm attitude to our efforts and the permission to use the name **DDscat.C++** for the new code.

Contents

1 Introduction 3

2 Quick start 4

2.1 Downloading the source code and examples . . . 4

2.2 Delivery overview . . . 4

2.3 Compiling the code . . . 5

2.3.1 MSVC . . . 5

2.3.2 Qt . . . 6

2.3.3 Mac OS . . . 6

2.3.4 Makefiles . . . 6

2.3.5 Precompiled binaries . . . 6

3 Running the application 6

3.1 Sequential version . . . 6

3.2 Parallel version . . . 7

4 Parameter file 7

4.1 Text parameter file . . . 7

4.2 Xml parameters file . . . 8

5 Test package 10

5.1 EllipsoN: N aligned homogenous isotropic ellipsoids . . . 11

5.2 AniEllN: N aligned homogenous anisotropic ellipsoids . . . 12

5.3 OctPrism: octagonal prism . . . 12

5.4 Octahedron: single isotropic octahedron particle . . . 13

5.5 Icosahedron: single isotropic icosahedron particle . . . 13

5.6 Dodecahedron: single isotropic dodecahedron particle . . . 15

6 General programming remarques 16

7 Target manager 17

7.1 TargetManager class . . . 17

7.2 AbstractTarget class . . . 18
7.3 LoadableTarget class . . . 19
7.4 How to add new target . . . 20

8 Solver engine **20**
8.1 AbstractSolver class . . . 21
8.2 How to add new solver . . . 21

9 Field engines and managers **22**
9.1 FFT engine . . . 22
9.2 Green function manager . . . 22
9.3 Dielectric manager . . . 22

10 Input and Output managers **22**

11 Restart manager **23**

12 Postprocessing **23**

13 Calltarget **23**

14 Finale **24**

References **25**

1 Introduction

Electromagnetic energy is scattered or absorbed by targets. It is an isolated grain (of arbitrary geometry and possibly with complex refractivity index) or 1-d or 2-d periodic structure of unit cells. According to discrete dipole approximation (DDA), the target is approximated with the array of polarizable particles (dipoles). The current version of the code works with electric dipoles only. Adding the magnetic properties is one of our next steps.

The theory of DDA and explanation of **DDSCAT** algorithms are given in [1] and in references therein. The current version of **DDSCAT** is 7.3.0 and we refer here this code as the parent one. The parent code User guide [1] is an appropriate and necessary book to start using the code.

Our code **DDscat.C++** is the **DDSCAT** rewritten in C++. Current version of **DDscat.C++** is a clone of the parent code but it contains some C++ specific features to make it easily modifiable and portable. We plan to extend the **DDscat.C++** functions in the future. It is open source and freely downloadable from `http://code.google.com/p/ddscatcpp/`. At the beginning of the story the idea was to have a good software for the students to study the photonics and IT in the single package. Step-by-step the code has changed and now we have the code with another design and architecture but mostly with the same functionality.

Like the parent code, **DDscat.C++** might be usable for many applications without modification. Anyway the users are encouraged to experiment with the code. Cloned C++ version, like the parent code, is distributed in the hope that the code will be useful.

If you publish results obtained using **DDscat.C++** , please:

- acknowledge the source of the code, make the citation of the paper [2] and mention the parent code by referencing the articles of B.Draine and P.Flatau [3] – [7],
- comply with the GNU General Public License: 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.

The author `Choliy.Vasyl@gmail.com` will be glad to read the mail from new **DDscat.C++** users or collaborators with your statement that you are new user. It will help in communication. Your reports of the bugs and errors in the code (if any) and your recommendations to the author are welcome.

The code and related data is collected here: `http://code.google.com/p/ddscatcpp/`. Please, visit it from time to time: we have a plan to keep the site up to current status of the code. We will keep some small bug-base there too.

2 Quick start

2.1 Downloading the source code and examples

DDscat.C++ is written in standard C++ with a little usage of STL. The style is quite plain and there are no extreme C++ features used. This means that the code should be portable to any platform with C++ compiler installed. We tested the distribution on Linux (Debian, Ubuntu), MacOS 10.5.8 where gcc is the preferable compiler, and on Windows with Microsoft VC 7.1 and Intel C++ 11 compilers. The code was tested on all those platforms with Qt 4.7.4 (it again uses gcc).

Normal (ordinal) makefiles are included in the distribution together with Microsoft VC 7.1, Qt 4.7.4 and Xcode projects. Please find your preferable (appropriate) IDE project and use it to compile the code.

The delivery consists of the single file `ddscatcpp.7.3.0.zip`. As Google does not provide Downloads, but only SVN or Git access to the code, please, find delivery at `space.univ.kiev.ua/Choliy/DDscatcpp/` or use SVN from Google code site.

The user may use precompiled binary files. A lot of them are presented in download section of the **DDscat.C++** Google code site.

2.2 Delivery overview

Select the empty directory, for example `ddscatcpp`, and unpack the delivery package there.

Explanation of the top level directory tree is given below:

.: project files for MSVC and the main makefile;

Bin: to store binary files generated with MSVC;

BinA: to store binary files generated with ordinal makefiles;

BinQt: to store binary files generated with Qt at any platform;

BinX: to store binary files generated with Xcode at Mac;

CallTarget: source files of CallTarget program;

CallTarget2: empty, reserved for Python code of CallTarget2 program;

DDscat: source files of **DDscat.C++** ;

DDscatQt: project files for Qt at any platform;

DDscatX: project files for Xcode at Mac;

diel: dielectric (and in the future - magnetic) property files;

Doc: this User guide;

Fourierlib: source files of FFT library;

General: source files of general kind;

Postprocess: source files of DDpostprocess program;

Processlib: source files of the library for Postprocess, Readnf1, Readnf2;

Readnf1: source files of Readnf1 program;

Readnf2: source files of Readnf2 program;

Results: scripts and parameter files for an extra examples, not included in the parent code distribution;

Solverlib: source files of the library of Conjugated Gradient solvers;

Targetlib: source files of Target library;

TestDDscat: testing subsystem (a little outdated);

Tests: scripts and parameter files to run parent code examples;

VtrConvert: source files of VtrConvert program (a little outdated);

Vtrlib: source files of the Vtr library;

Xml: Xml related stuff.

2.3 Compiling the code

The compilation of the code is essential for any platform. Open appropriate project file and select Build All from the main menu of your IDE. We made serious efforts to make the code compilable without warnings. Warnings during the compilation of the code should be interpreted as if something went wrong.

2.3.1 MSVC

The solution file `DDscat.sln` resides in main directory. It was created with MSVC 7.1. Select Build – Rebuild All from the main menu. Compilation results will be collected in `Bin` directory. Binary xml distribution files should be copied in `Bin` before build.

2.3.2 Qt

The main project file `DDscatQt.pro` resides in `DDscatQt` directory. It was created with Qt Creator 2.6.1 and Qt 4.7.4. Select Build – Rebuild All from the main menu. Compilation results will be collected in `BinQt` directory. Binary xml distribution files should be copied in `BinQt` before build.

The user can use Qt project to build **DDscat.C++** under Linux or Mac OS X. Some additional work is necessary to replace pathes and library extensions in `*.pro` files.

2.3.3 Mac OS

For Xcode users we provide Xcode project files in `DDscatX` directory. It was created with Xcode 3.0 at Mac OS 10.5.8. The main project file is `DDscat.xcodeproj`. Select Rebuild from main Xcode directory. Compilation results will be collected in `BinX` directory. Binary xml distribution for Mac OS X 10.5.8 is already installed as a system component. Anyway, the project is configured as if binary xml files are already present in `BinX` before build.

2.3.4 Makefiles

For Linux users it is essential to have the distribution based upon the autoconf tool. Despite of that we provide just an ordinal makefiles, stored in every directory of the distribution. Autoconf is planned for the future releases. Just type `make` in a console opened at the main directory location and gcc (if exists) will produce the distribution in `BinA` directory. Binary xml distribution files should be copied in `BinA` before make.

2.3.5 Precompiled binaries

The user can download all binaries stored in a single `zip` file from authors web site `space.univ.kiev.ua/Choliy/DDscatcpp/`. The files contain all necessary binaries including xml libraries. These files are copies of `Bin*` directories. Filenames are `DDscatcpp.7.3.0.MSVC.zip`, `DDscatcpp.7.3.0.WinQt.zip`, `DDscatcpp.7.3.0.Xcode.zip`.

3 Running the application

3.1 Sequential version

After the successfull compilation go to an appropriate `Bin` directory where all necessary binary files should be already collected bu build process. The user should identify `DDscat`, `Readnf1`, `Readnf2`, `CallTarget`, `VtrConvert`, `DDpostprocess` executive files and

Fourierlib, Solverlib, Targetlib, Vtrlib libraries and third party libraries libXml2, iconv, zlib1.

DDscat.C++ may be run with the single parameter (the name of par file) or without parameters. In the latter case the **DDscat.C++** executive search the current directory for ddscat.par file. If the file is not found, the executive will search for ddscatpar.xml and use it.

DDscat.C++ generates a lot of messages in stdout (file for messages, normally attached to display). We recommend to redirect stdout to some log file to analyse it in post-run mode. Error messages are written into stderr (error file, normally attached to display too) which is better to leave on screen.

3.2 Parallel version

All the parallel code in **DDscat.C++** 7.3.0 is temporary disabled. MPI and OpenMP codes will stay disabled until version 7.3.2 while the CUDA - based code will be released in 7.3.1 after all tests. MPI and OpenMP codes are present in C++ code but are absolutely not tested. User might catch unpredictable errors (as a minimum, a lot of compiler messages) if tries to use MPI or OpenMP with current version of **DDscat.C++** .

4 Parameter file

The **DDscat.C++** may be controlled with parameters file of the parent code but some additional freedom in the parameter file is allowed. There are two special compositions: Water and Ice allowed as the composition file names. These are reimplementation of the refwat and refice routines from **DDSCAT** 6.

4.1 Text parameter file

All string parameters may be presented without putting into apostrophes (if they do not contain blank characters). So, 'GPFAFT' like in the parent code and GPFAFT are identical and are allowed. All the lines starting with a) the apostrophe and the blank or b) the apostrophe followed with three asterisks or c) the exclamation sign are interpreted as an comments and are just skipped. The user may add as many of such lines as he need, for example for documenting reasons.

The target name may be any length single word with free capitalization and all underscore symbols ignored by **DDscat.C++** . That is why SPH_ANI_N and SphaniN or even S__p_HAn___iN__ are identical and are allowed.

DDscat.C++ makes memory allocation only once during the target loading. That is why 8th and 9th lines of the parameter file are ignored but should be present in the file.

In the definition of composition files after line 13 there might be a lot of file names given in the parameter file. **DDscat.C++** allows the usage of equality sign after some amount of composition files given. It means that all already given file names will be cyclically repeated until their amount become equal to `NCOMP`.

For example there is a portion of a normal par file:

```
>>>> begin
12         = NCOMP = number of dielectric materials
‘../diel/m1.33_0.01’ = file with refractive index 1
‘../diel/m1.50_0.01’ = file with refractive index 2
‘../diel/m1.50_0.02’ = file with refractive index 3
=
>>>> end
```

This means that there are 12 composition files in the example and composition 4 is equal to composition 1, composition 5 to 2, and so on until composition 12 (obviously equal to 9, then equal to 6 then equal to 3). The error message is generated when amount of the given file names exceeds `NCOMP`.

4.2 Xml parameters file

DDscat.C++ can be controlled with xml parameter files. DTD for xml parameter files `DDscatcpp.dtd` resides in `Xml` subdirectory. An example `xml` parameter file for `Rctglprsm` test from **DDSCAT** is given below. Line numbers the every line starts with are not the part of an `Xml` file and are present for the orientation. Users are allowed to add any amount of comment lines in the `Xml` parameter file. Just start them with `<!--` and end with `-->`, see lines 11 and 12 for example, so multiline comments are allowed.

```
 1   <?xml version="1.0" encoding="utf-8"?>
 2   <!DOCTYPE DDScatParameterFile SYSTEM "ddscatcpp.dtd">
 3   <DDScatParameterFile ver="7.3">
 4   <Preliminaries>
 5      <Cmtorq Value="NOTORQ"/>
 6      <Cmdsol Value="PBCGS2"/>
 7      <CmdFFT Value="GPFAFT"/>
 8      <Calpha Value="GKDLDR"/>
 9      <Cbinflag Value="NOTBIN"/>
10   </Preliminaries>
11   <!--NCOMP number of dielectric materials
```

```
    DIELEC file with refractive index 1-->
<TargetGeometryAndComposition>
   <Cshape Name='Rctglprsm'/>
   <Shpar Pos="1" Value="16" Comment="x size of the target"/>
   <Shpar Pos="2" Value="32"/>
   <Shpar Pos="3" Value="32"/>
   <Ncomp Amount='1'>
      <Dielec Pos="1" File="../diel/Au_evap"/>
   </Ncomp>
</TargetGeometryAndComposition>
<NearfieldCalculation Nrfld="0">
   <Extendxyz Xm="0.0" Xp="0.0" Ym="0.0" Yp="0.0" Zm="0.0" Zp="0.0"/>
</NearfieldCalculation>
<Tol Value="1.00e-5"/>
<Mxiter Value="300"/>
<Gamma Value="1.00e-2"/>
<Etasca Value="0.5"/>
<VacuumWavelengths First="0.5000" Last="0.5000" HowMany="1" How="LIN"/>
<Nambient Value="1.000"/>
<Aeff First="0.246186" Last="0.246186" HowMany="1" How="LIN"/>
<IncidentPolarization Iorth="2">
   <PolarizationState>
      <X Re="0" Im="0"/>
      <Y Re="1" Im="0"/>
      <Z Re="0" Im="0"/>
   </PolarizationState>
</IncidentPolarization>
<Iwrksc Value="1"/>
<PrescribeTargetRotations>
   <Beta  Min="0." Max="0." Number="1"/>
   <Theta Min="0." Max="0." Number="1"/>
   <Phi   Min="0." Max="0." Number="1"/>
</PrescribeTargetRotations>
<SpecifyFirst Iwav="0" Irad="0" Iori="0"/>
<S_ijMatrix Number="6">
   <ij Value="11 12 21 22 31 41"/>
</S_ijMatrix>
<ScatteredDirections Cmdfrm="LFRAME" Nplanes="2">
   <Plane N="1" phi="0."  MinThetan="0." MaxThetan="180." Dtheta="5"/>
   <Plane N="2" phi="90." MinThetan="0." MaxThetan="180." Dtheta="5"/>
</ScatteredDirections>
```

```
53   </DDScatParameterFile>
```

There should not be difficulties in understanding of the `xml` parameter file. Most of its lines are quite self-explanatory and are easily mapped onto the lines of the text par file. Presented example contains all possible tags. Sometimes, when there is no necessity, some of the tags may be dropped. For example if the user does not plan to do nearfield calculations one can omit lines 22 - 24, or 27th if there is no gamma used in calculations, etc. Comment attribute in `Shpar` element may be dropped too.

5 Test package

The `Tests` and `Results` directories contain scripts and parameter files for all parent code examples and all targets mentioned in **DDSCAT** User guide [1]. The `Tests` parameter files are identical to those of **DDSCAT** User guide, but `Results` ones are quite artificial and should be used only for illustration. Main difference between `Tests` and `Results` is that all results do nearfield calculation and have MayaVi2 snapshots.

Any test or result resides in its own direstory. The directory contains par and xml parameter files together with target explanation files (targ) if any.

To run tests or results go into appropriate directory and run `RunAll*` script. The script run **DDscat.C++** for all subdirectories and then run `Readnf1` and `Readnf2` for them to produce `vtr` files for `MayaVi2` and field crossing along the line. All scripts are quite elementary.

To add new Result to the `Result` directory one should:

- create the own directory `DirectoryName`,
- copy `RunResult.bat`, `RunReadnf1.bat` and `RunReadnf2.bat` into it from any subdirectory, copy extra files, like target explanation file into it,
- modify them if you need some extra files,
- run `TheResult DirectoryName` script,
- run `TheReadnf1 DirectoryName` script if you need near target field,
- run `TheReadnf2 DirectoryName` script if you need field crossing along the line,
- find resulting files and logs in `DirectoryName`.

Please, refer to **DDSCAT** User guide [1] for target explanations. The only targets explained here are new ones. In `Tests` and `Results` directories one can find all necessary things to run the tests and results including the MayaVi2 snapshot from our runs.

5.1 EllipsoN: N aligned homogenous isotropic ellipsoids

The target consists of N ellipsoids identical in size but possibly different in composition placed along x axis. There are 5 parameters:

- SHPAR_1 = length of the ellipsoid in x direction;
- SHPAR_2 = length of the ellipsoid in y direction;
- SHPAR_3 = length of the ellipsoid in z direction;
- SHPAR_4 = N - number of ellipsoids;
- SHPAR_5 = distance between ellipsoids surfaces along x direction.

User should provide N compositions and set `NCOMP = N`.

The portion of example calculation of the `par` file is copied just below with MayaVi2 visualization of the electric field on Fig. 1:

```
'ELLIPSON' = CSHAPE*9 shape directive
24. 36. 30. 5. 6. = shape parameters 1 - 5
5          = NCOMP = number of dielectric materials
'../diel/m0.96_1.01' = file with refractive index 1
'../diel/m0.96_1.01' = file with refractive index 2
'../diel/m0.96_1.01' = file with refractive index 3
'../diel/m0.96_1.01' = file with refractive index 4
'../diel/m0.96_1.01' = file with refractive index 5
'**** Additional Nearfield calculation? ****'
1 = NRFLD (=0 to skip nearfield calc., =1 to calculate nearfield E)
0.1 0.1 0.5 0.5 0.5 0.5 (fract. extens. of calc. vol. in -x,+x,-y,+y,-z,+z)
```

As we already mentioned above if `par` file contains a lot of composition files, they can be replaced with equality sign, that is why the portion of the `par` file can be replaced with:

```
'ELLIPSON' = CSHAPE*9 shape directive
24. 36. 30. 5. 6. = shape parameters 1 - 5
5          = NCOMP = number of dielectric materials
'../diel/m0.96_1.01' = file with refractive index 1
=
'**** Additional Nearfield calculation? ****'
1 = NRFLD (=0 to skip nearfield calc., =1 to calculate nearfield E)
0.1 0.1 0.5 0.5 0.5 0.5 (fract. extens. of calc. vol. in -x,+x,-y,+y,-z,+z)
```

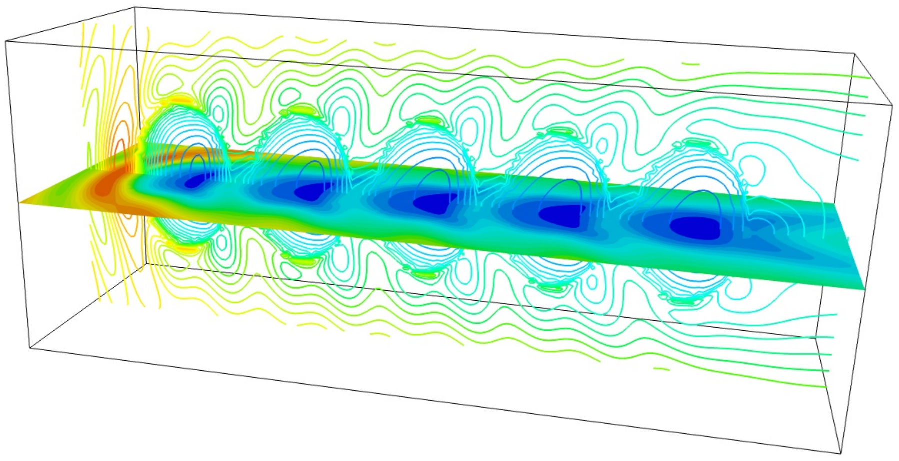

Figure 1: General view of the electric field in EllipsoN result.

5.2 AniEllN: N aligned homogenous anisotropic ellipsoids

The target consists of N anisotropic ellipsoids identical in size but possibly different in composition placed along x axis. There are 5 parameters identical to those of `ELLIPSON`. User should provide 3*N compositions and set `NCOMP = 3*N`.

The portion of example calculation of the `par` file is copied just below with MayaVi2 visualization of the electric field on Fig. 2:

```
'ANIELLN' = CSHAPE*9 shape directive
24. 36. 30. 5. 6. = shape parameters 1 - 5
15          = NCOMP = number of dielectric materials
'../diel/m1.33_0.01' = file with refractive index 1
'../diel/m1.50_0.01' = file with refractive index 2
'../diel/m1.50_0.02' = file with refractive index 3
=
'**** Additional Nearfield calculation? ****'
1 = NRFLD (=0 to skip nearfield calc., =1 to calculate nearfield E)
0.1 0.1 0.5 0.5 0.5 0.5 (fract. extens. of calc. vol. in -x,+x,-y,+y,-z,+z)
```

5.3 OctPrism: octagonal prism

This target represent the single octagonal prism particle. It is mostly the same as hexagonal prism, but only one orientation (main prism axis lies along the x coordinate) is provided

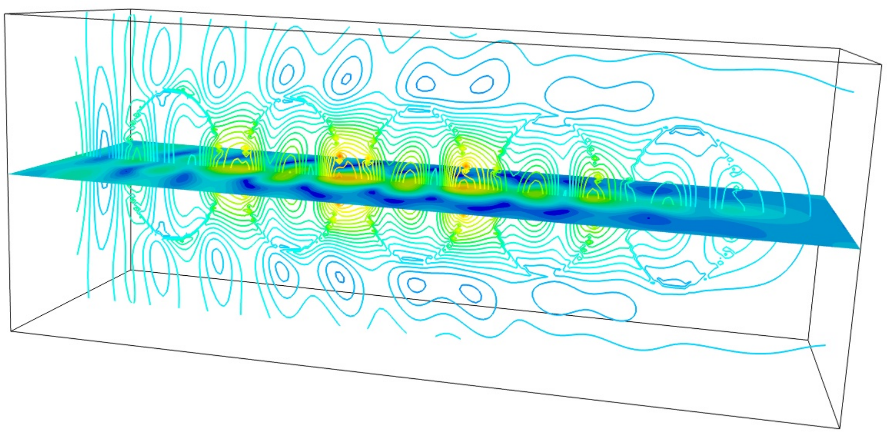

Figure 2: General view of the electric field in AniElIN result.

now. There are two parameters:

- SHPAR_1 = length of the prism in x direction;
- SHPAR_2 = distance between opposite vertices of one octagonal face.

User should provide 1 composition and set NCOMP = 1. Target name is OCTPRISM.

Figure 3 represents the electric field near the octagonal prism target with parameters equal to 24. and 30.

5.4 Octahedron: single isotropic octahedron particle

The target is octahedron particle. Each line that pass through two oposite vertex is parallel to the x,y, and z axis, respectively. The target needs the only parameter: a distance between two opposite vertices, or a diameter of escribing sphere. The target is isotropic, so NCOMP = 1 and the user should provide only one composition file. Figure 4 represents the electric field near the octahedron target with parameter equal to 40. Target name is OCTAHEDRON.

5.5 Icosahedron: single isotropic icosahedron particle

The target is icosahedron particle oriented like the octahedron one. The target needs the only parameter: the lenght of the edge. The target is isotropic, so NCOMP = 1, and the user should provide only one composition file. Figure 5 represents the electric field near the icosahedron target with the parameter equal to 24. The target name is ICOSAHEDRON.

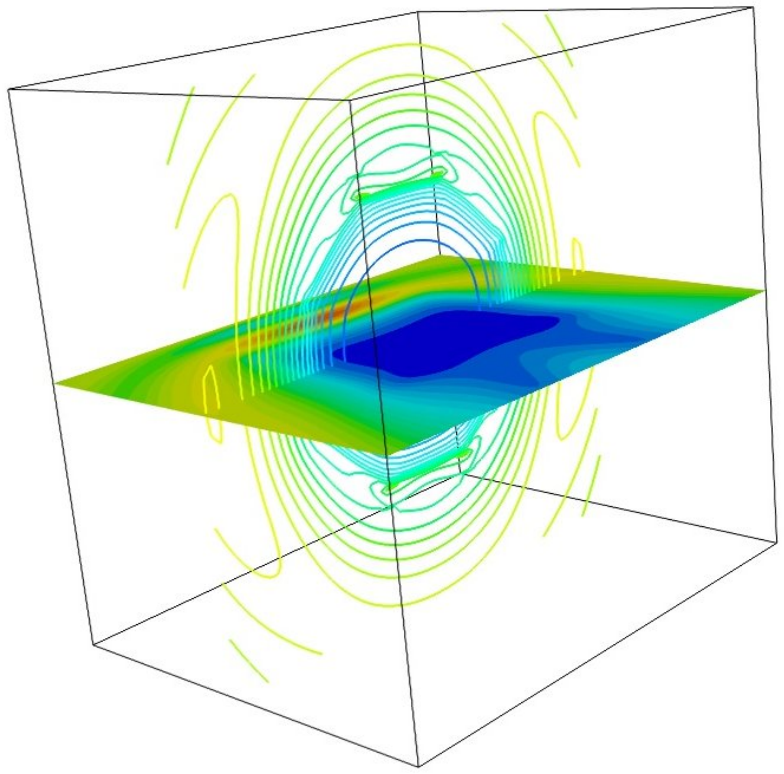
Figure 3: General view of the electric field in OctPrism result.

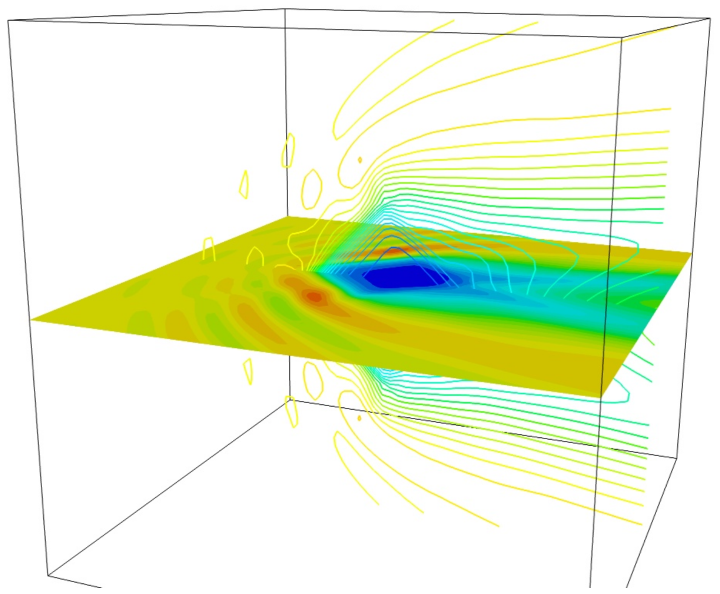
Figure 4: General view of the electric field in Octahedron result.

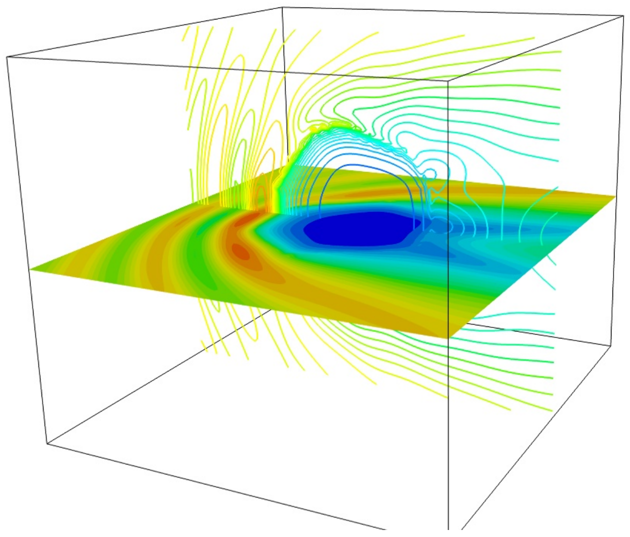
Figure 5: General view of the electric field in Icosahedron result.

5.6 Dodecahedron: single isotropic dodecahedron particle

The target is dodecahedron particle. The pentagonal base is parallel to the xy plane and one edge of the base is parallel to the y axis. The target needs the only parameter: the lenght of the edge. The target is isotropic, so `NCOMP = 1`, and the user should provide only one composition file. Figure 6 represents the electric field near the dodecahedron target with the parameter equal to 24. The target name is `DODECAHEDRON`.

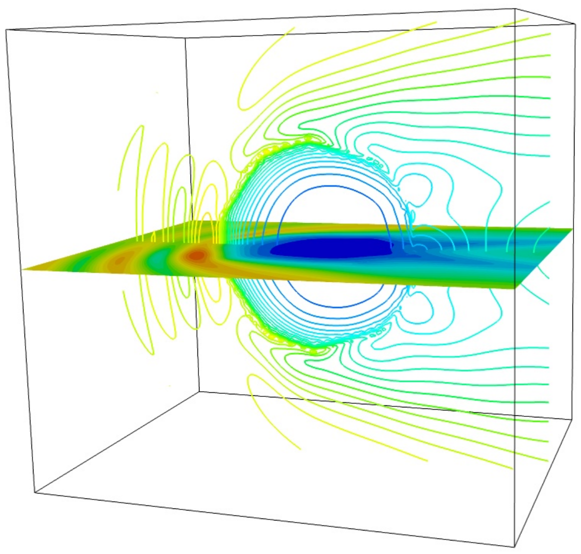

Figure 6: General view of the electric field in Dodecahedron result.

6 General programming remarques

The code consists of components with strictly defined communication protocol and lightweight replacement, modification and refactoring (plugin paradigm). **DDscat.C++** is a set of dynamically linkable libraries with defined and fixed communication interfaces.

Examples and testing capabilities are an essential part of the code. As running of all tests consumes a lot of time, we don't use CppUnit but our own code and scripts library to be run on request. All **DDSCAT** tests work fine as a part of **DDscat.C++** code.

The overall view of the architecture and its main blocks are presented on Fig. 7. The users familiar with the parent code may easily identify known code blocks. The asterisk as an upper index marks new code parts, introduced in C++ version. Every code portion is controlled with and is communicated via the specially designed manager components. These code snippets are singletons.

The code uses C-style of indexing. This means that all indexes in all arrays in the code always starts from zero, not one, as it is in fortran.

The code and most of the names are case sensitive, but target names are not. Please, be on guard with it.

The code needs cosmetics and refactoring. Strictly speaking the code is not written in C++, much better to say that the code is written in C with some amount of classes. That is why the code mostly does not contain STL. It is our permanent plan for the future releases: refactoring to add STL and make cosmetics in necessary places but without fanatism.

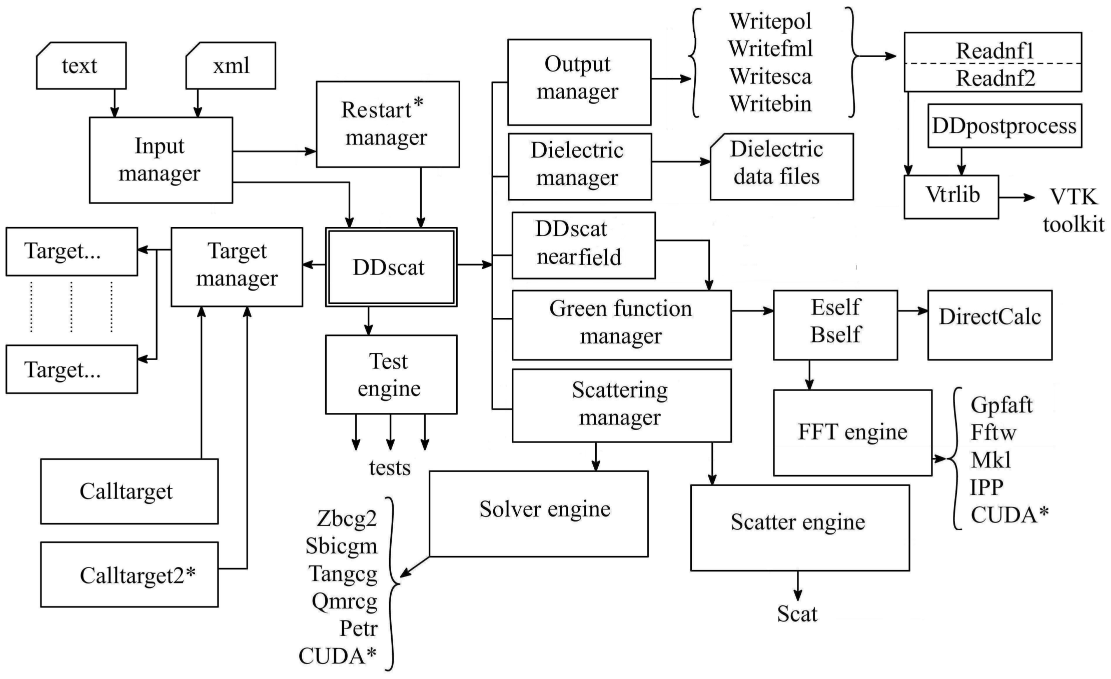

Figure 7: General view of the code architecture.

7 Target manager

Target manager manipulates the targets. The target explains the grains geometry or represents elementary cell to build 1-d or 2-d infinite periodic arrays of targets. The parent code contains a lot of different geometries already implemented. These are ellipsoids (spheroids), prisms, cylinders, disks, slabs, tetrahedra, possibly with holes and their simple joints. Some of the targets are just a combination or multiplications of existing ones.

7.1 TargetManager class

This singleton class controls life cycle of the current target and owns information about it. It is build as a class factory with the possibility to self register the target in the factory.

To access and register itself in the factory the target should be accompanied with `REGISTER_TARGET` macro. The parameters of the macro are:

- the name of the target, an ordinal word with any capitalization;
- the number of target parameters including file name if any;
- will the target use additional file, false or true boolean value;
- the position of periods in parameters if the target is periodic, otherwise -1;

- the number of required composition, 0 - doest'n matter, not 0 - the number of required ncomp in par file;
- free description of the target in one line.

For example, lets register `Beautiful_Particle` as not periodic target with two parameters including the file name, three compositions required. The macro for the target is:

```
REGISTER_TARGET(Beautiful_Particle,2,true,-1,3,"Flower-like particle")
```

The macro should be placed somewhere in cpp file of the target class, whose name should be `Target_Beautiful_Particle`. The macro forces the user to add two mandatory functions to the code, namely `void Target_Beautiful_Particle::SayHello(FILE *stream)` and `const char *TargetVerboseDescriptor_Beautiful_Particle(int num)`.

The first function is used to write the values of the target parameters into the stream. The second one returns string representation of the parameter identified with num value. All of the targets in delivery have those functions. Please, use the code for additional information.

7.2 AbstractTarget class

This class is the core of the TargetManager library. Despite of its name the class does not contain pure virtual functions. It is just a good name for the class, capable to represent any possible target.

The class contains:

- nat - int - the total amount of dipoles (places for dipoles);
- nat0 - int - the total amount of occupied dipoles;
- nx, ny, nz - int - the dimension of the target;
- ixyz[nat0, 3] - int - relative coordinates of occupied dipoles;
- minJx, maxJx, minJy, maxJy, minJz, maxJz - int - min and max limits of ixyz values;
- icomp[nat, 3] - short - a dipole composition;
- ncomp - int - the number of compositions;
- iocc[nat] - bool - the dipole occupancy sign;
- shpar - real - target parameters;
- pyd, pzd - real - periods if the target is periodic;

- ianiso - the isotropic flag;
- a1, a2, dx, x0 - known vectors.

The most valuable method of AbstractTarget in Build, which just calls a lot of another methods:

```
void AbstractTarget::Build(void)
{
    Sizer();                // determines the nx,ny,nz and min/max values of ixyz
    Descriptor();           // creates target descriptor string
    Allocator();            // allocates ixyz, iocc, icomp
    Vector();               // defines a1 and a2 vectors
    VectorX();              // defines x vector
    Printer();              // prints target head and creates target.out file
    ShiftDipolesAndX();     // shift x and ixyz to have the first dipole at (1,1,1)
    PrepareIaniso();        // recognizes if the target is anisotropic or not
    PreparePyzd();          // prepares periodicities if necesary
}
```

All the functions from `Sizer` to `PreparePyzd` are dummy in AbstractTarget class. The user should create its own versions if added a new target to the TargetManager.

Two of the functions, `Sizer` and `Allocator` work together to allocate the memory for the target in a single job. `Sizer` determines min and max values of the dipole coordinates and nx, ny, nz - sizes of the cubic cell to insert the target in, but it does not do the allocation. In most cases it does a dummy allocation - everything is doing as if it is an allocation but the only min/max values are determining. As if cell dimensions are determined, the `Allocator` really allocates the memory and put the dipole data in.

The main task of the user when added new targets is to place the dipole x, y, z into ixyz and then using `GetLinearAddress` to determine the correct place of composition data in the iocc array. GetLinearAddress needs correct min/max values. **All arrays except specially stated use C-like indexing method.**

7.3 LoadableTarget class

This class is inherited from AbstractTarget and is designed to represent the target with some data in additional (loadable during preparation) targ file. There is only one additional function in LoadableTarget class `void LoadableTarget::Reader(void)` which is used to load the targ file and add the information from it to the target.

7.4 How to add new target

This is step by step receipt to add a new target to the TargetManager. We will do it using an `OctPrism` target name (but in general sence) as an example. The target name should be unique in the **DDscat.C++** scope.

Every target is represented with its class, but having in mind possible future extensions and modifications of the code it is better to add two classes: the generic one to represent the properties of the prisms, and the concrete one to represent the properties of the octagonal prism. Namely in the case of prisms, we already have a `Tarhex` class as a generic one and a `Target_HexPrism` as a concrete one. Obvious future refactoring should result in something like

```
Prism -> Tarhex -> Target_HexPrism
      -> Taroct -> Target_OctPrism
```

but it needs some additional efforts.

1. Edit the `Targetlib\TargetDefinitions.h` file and add a string `TargetType_OctPrism` somewhere in the definition of `enum TargetType`.
2. Add a new generic class `Taroct` to TargetManager inheriting it from `AbstractTarget` or from `LoadableTarget` if the new target will read some extra information from a targ file.
3. Add a new concrete class `Target_OctPrism` to TargetManager inheriting it from just added generic class `Taroct`. The concrete class name should always be `Target_` followed with a preselected target name (`OctPrism` now).
4. Add a `REGISTER_TARGET` macro somewhere in a cpp file.
5. Add a `const char *TargetVerboseDescriptor_OctPrism(int num)` function to a cpp file.

Now the user should implement the functions, called from `AbstractTarget::Build` (see AbstracTarget class above), and Reader if the target is loadable.

8 Solver engine

Solver is a generic name for the code components designed to solve sets of linear equations. The current version of **DDscat.C++** uses only Conjugated gradient (CG) solvers from CGPACK library of P.Flatau[1]. These solvers are well explained in **DDSCAT** User Guide

[1]http://code.google.com/p/conjugate-gradient-lib/

[1]. For **DDscat.C++** it is not mandatory to use CG codes. The user are allowed to add own solvers.

8.1 AbstractSolver class

Singleton AbstractSolver class plays two main roles. It is solver factory and solver manager simultaneously. The class object controls the lifecycle of the solver and manages the access to the solver in use.

To access and register itself in the factory the solver should be accompanied with a `REGISTER_SOLVER` macro. The parameters of the macro are:

- the name of the class the solver is stored in;
- the string which represents the solver for the user.

The last string to be used for identifying the solver in a par file.

For example, lets register `BeautifulSolver`. The macro for the solver is:

```
REGISTER_SOLVER(BeautifulSolver,"BEAUTY")
```

8.2 How to add new solver

This is step by step receipt to add new solver to the Solver engine. We will do it using Beauty solver name as an example. The solver name should be unique in the **DDscat.C++** scope.

- Edit `enum SolMethod` in `General\Enumerator.h` and add `SolMethod_Beauty` string before `SolMethod_End`;
- edit functions `SolEnumerator` in `General\Enumerator.cpp` and add strings for the new solver;
- add the new class `Solver_Beauty` to Solverlib subproject, inheriting it from `AbstractSolver` class;
- add a macro `REGISTER_SOLVER` somewhere in a cpp file.

Every solver is represented with its class. The user is allowed to name the class members in any useful way, but there should be:

- the solver parameters, they should be set with the special function `SetParameters` before the call of the solver function;

- the solver function with two mandatory parameters: the initial guess vector and a right side vector and possibly additional parameters; in the future refactoring we plan to retain only first two parameters;
- the external function Matvec which is used by the solver to do matrix-vector multiplications; Matvec is set to the solver with the `SetMatvec` function.

9 Field engines and managers

9.1 FFT engine

This engine is the another singleton which is used to manage the access to FFT codes. The current version of **DDscat.C++** is capable to work only with the Gpfaft code of C.Temperton [8]. Usage of FFTW and Intel routines are temporary disabled (to 7.3.1).

9.2 Green function manager

Green function manager organizes direct calculation of electric and magnetic field with Green function approach. In current **DDSCAT** there is a lot of code duplicates around `Eself` and `Bself` routines, which are very similar. In **DDscat.C++** there is a `Subsystem` class which is (or may be) instantiated for electric (Eself) and/or magnetic (Bself) calculations. That saves a lot of memory and leeds to greater targets.

Green function manager effectively manipulates with the instances of `Subsystem` class and replaces the calculations with data exchange whenever possible.

9.3 Dielectric manager

Dielectric manager is a little self-made data management class. It controls dielec files, stores dielectric data, provides the access to dielectric values. Our future plans include introducing the magnetic dipoles into calculations. Dielectric manager is already able to manipulate magnetic data too.

10 Input and Output managers

Input manager manages **DDscat.C++** parameters, which are organized in the `DDscatParameters` class. It reads par or xml files and provides the access to the data stores there.

Output manager is responsible for preparing the output files. It comprises all `Write*` routines from **DDSCAT** and manipulates the calculation results to create pretty-looking output.

11 Restart manager

Restart manager is in testing phase and will be released in version 7.3.2. Main task of Restart manager is to do a restart of the calculation without loss of the data after sudden events, for example an electricity failure.

12 Postprocessing

In the current version (7.3.0) **DDscat.C++** includes standalone programs:

- Readnf1 - to create vtr files for vizualization of the electric field with MayaVi2 software;
- Readnf2 - to make a field crossing along the specified line;
- Postprocess - component appeared in **DDSCAT** 7.3.0 which do the jobs of both Readnf's.

Readnf1 is controlled with a quite elementary par file:

```
'w000r000k000.E1'             = name of file with E stored
'VTRoutput'                   = name of VTR output file
1   = IVTR  (set to 1 or 2 to create VTR output with |E| or |E|^2)
```

the same is for Readnf2:

```
'w000r000k000.E1'            = name of file with E stored
1   = ILINE (set to 1 to evaluate E along a line)
-2.0 0.0 0.0 2.1 0.0 0.0 501  = XA,YA,ZA, XB,YB,ZB (phys units), NAB
```

The par files of those components are the same as in **DDSCAT** 7.2.2, they are just subdivided into two parts. The par file for `Postprocess` is the same as for Readnf of **DDSCAT** 7.2.2.

13 Calltarget

Function of CallTarget is explained in Fortran version User Guide. CallTarget2 is a new wxPython component to be released with **DDscat.C++** 7.3.2 and will help users to create the targets interactively.

14 Finale

This User Guide is a subject of permanent changes and will stay improving together with **DDscat.C++** code.

The source code and some additional information are available at Google code site: `http://code.google.com/p/ddscatcpp/`

Post your bugs and suggestions via E-mail to the author `Choliy.Vasyl@gmail.com`. Please, provide your E-mail addresses and identify yourself as a user or a tester or a hacker or ... of the code. As if the author will be able to inform the engaged persons with news, bug fixes, new releases.

Extension of the Targetlib with new targets are welcome first.

If you plan to use **DDscat.C++** please, cite the article

```
Choliy V. 2013, "The discrete dipole approximation code DDscat.C++:
features, limitations and plans".
Adv.Astron.Spa.Phys., 3, 3-10
```

and the articles of B.Draine and P.Flatau, referenced in Finale of **DDSCAT** User Guide [1].

The **DDscat.C++** author appreciates receiving copies of the articles where **DDscat.C++** is mentioned.

Great thanks to B.Draine and P.Flatau for positive and warm attitude to our efforts and the permission to use the name **DDscat.C++** for the new code.

References

[1] Draine, B.T., and Flatau, P.J. 2013, User Guide for the Discrete Dipole Approximation Code DDSCAT 7.3, `http://arxiv.org/abs/xxxx.xxxx`

[2] Choliy, V. 2013, "The discrete dipole approximation code DDscat.C++: features, limitations and plans". `Adv.Astron.Spa.Phys., 3, 3-10.`

[3] Draine, B.T., 1988. "The Discrete-Dipole Approximation and its Application to Interstellar Graphite Grains". `Astrophys. J., 333, 848-872.`

[4] Goodman, J.J., Draine, B.T., & Flatau, P.J., 1990. "Application of fast-Fourier transform techniques to the discrete dipole approximation". `Optics Letters, 16, 1198-1200.`

[5] Draine, B.T. & Flatau, P.J., 1994. "Discrete-dipole approximation for scattering calculations". `J.Opt.Soc.Am., 11, 1491-1499.`

[6] Draine, B.T. & Flatau, P.J., 2008. "Discrete dipole approximation for periodic targets: I. Theory and tests". `J.Opt.Soc.Am., 25, 2693-2703.`

[7] Flatau, P.J. & Draine, B.T., 2012. "Fast near-field calculations in the discrete dipole approximation for regular rectilinear grids". `Optics Express, 20, 1247.`

[8] Temperton C., 1992. ""A Generalized Prime Factor FFT Algorithm for any $N = 2^p 3^q 5^r$" `SIAM J.Sci.Stat.Comp. 13, 676.`