DDscat.C++ 7.3.1

User and programmer guide

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

Last revised 26 February 2017

Abstract

DDscat.C++ 7.3.1 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. New version, introduced in this document has list of aprovements.

Despite the fact that **DDscat.C++** in its current version is a little bit more than just a clone, usage of **DDscat.C++** for electromagnetic calculations is the same as of **DDSCAT** . Please, refer to the latest "User Guide for the Discrete Dipole Approximation Code DDSCAT ...", for example [1] to start using the code(s). **DDscat.C++** code will follow the releases of **DDSCAT** and will implement all new features and possibilities implemented there.

This document introduces new features of **DDscat.C++** and enhancements of the new code. To understand how the code works one should familiarize oneself with User guides of previous C++ version [2] and the latest Fortran one [1].

The author is grateful 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.

Contents

1 Introduction **3**

2 Target manager **5**

2.1 Targets are constructible 5

2.2 Elementary targets 5

2.2.1 Orientable targets 6

2.2.2 Absolute targets 8

2.2.3 Calculable target 10

2.3 Target manipulation 10

2.4 Does the dipole belong the target 12

2.5 Gallery 15

2.6 BNF of target language 18

3 Parameter file **20**

4 Solver engine **21**

4.1 AbstractSolver class 21

4.2 How to add new solver 21

5 Field engines and managers **22**

5.1 FFT engine 22

5.2 Green function manager 22

5.3 Dielectric manager 23

6 Input and Output managers **23**

7 Postprocessing **23**

8 Calltarget and DdaCovnert **24**

9 Finale **24**

References **26**

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 the discrete dipole approximation (DDA), a target is approximated with an array of polarizable particles (dipoles). The current version of the code works with electric and magnetic dipoles. Magnetic part of the code was introduced starting from the ideas of **DDscat-mu** code [10], but it was subjected to significant changes to fit existing architecture of **DDscat.C++** . In current release the magnetic properties of the dipoles inserted in the code but still need extensive testing.

The theory of the DDA and explanation of **DDSCAT** algorithms are given in [1] and in references therein. The current version of **DDSCAT** is 7.3.2 (as on February, 2017) and we refer here this code as the parent one. The parent code User guide [1] and User guide of current C++ version [2] are appropriate and necessary documents to start using the code.

Our code **DDscat.C++** is the **DDSCAT** rewritten in C++. Despite of that the current version of **DDscat.C++** is not just a clone of the parent code. It contains not only some C++ specific features to make it easily modifiable and portable, but a list of enhancements to the architecture to make the code faster.

We plan to extend the **DDscat.C++** functions in the future and keep our version as close as possible to the parent code. It is open source and freely downloadable from `http://space.univ.kiev.ua/Choliy/DDscatcpp/` along with User guide. Former version(s) is still downloadable too.

Like the parent code, **DDscat.C++** might be usable for many applications without any modifications. Anyway the users are encouraged to experiment with the code. 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 [3] and mention the parent code by referencing the articles of B.Draine and P.Flatau [4] – [8],
- 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 with 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 accessible via authors personal page only (we are sorry, but Google code has stopped): `http://space.univ.kiev.ua/Choliy/`. Please, visit it from

time to time: we have a plan to keep the site up to current status of the code.

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 in Fig. 1. 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.

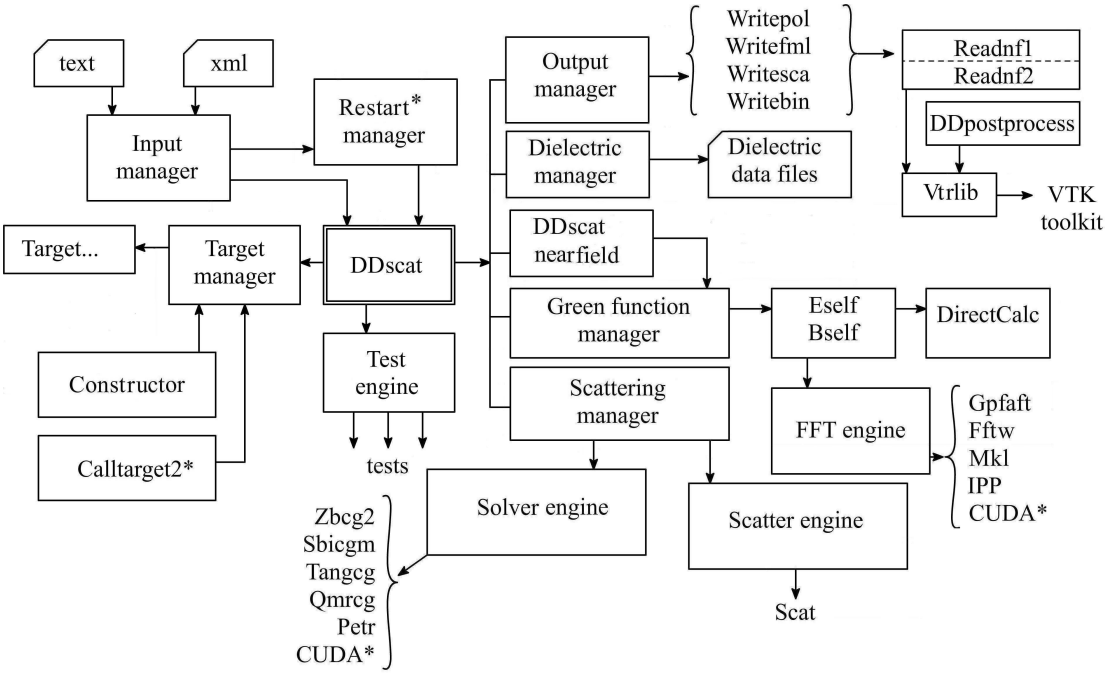


Figure 1: General view of 7.3.1 C++ code architecture.

The code uses C-style of indexing. This means that all indices, including the array element ones, in the code always starts from zero, not one, as it is in fortran.

The code and most of the names are case sensitive.

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. It is our permanent plan for the future releases: refactoring, refactoring again and cosmetics in necessary places but without using the newest version of the C++ standards.

2 Target manager

Target manager[(1)] 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), pyramids, prisms, cylinders, disks, slabs, tetrahedra, possibly with holes and their simple joints. Some of the targets are just a combination of the existing ones.

C++ code starting from version 7.3.1 uses a new approach to the construction of targets. Classes for Target manager and all elementary targets have been rewritten from scratch. General methodology of target manipulation has changed too.

2.1 Targets are constructible

New approach is represented by `ConstructTarget` class and user preselected target type. User creates targets on the fly, inserting the creating instruction for new target directly into parameter file. To do this, user should name new target in a parameter file by any single word. This string is used only to name somehow the target and to keep the general format of the parameter file not changed. The built target is saved in `target.out` file and in VTK readable format, like `VtrConvert` does. The next line (or lines) of the par file should contain the phrase, explaining the target to be created. This phrase should ends with semicolon ';' symbol. See below about it in details.

There is a component **Calltarget** retained from parent code but totally changed. Now it uses parameter file with the first line containing the name of the generated target and the target explanation in the next line(s). Semicolon symbol at the end of input is mandatory in any case. Announced earlier `CallTarget2` as a visual target constructor will appear in the next releases.

2.2 Elementary targets

The target should be created as a combination of items from the library of elementary targets. Each elementary target has its own coordinate triad, position of its center and direction of the axes. External coordinate triad XYZ is rectangular and is right oriented. In terms of parent code it is the laboratory frame.

We use term `base` in the sense "the circle is the base of the cone", if it is essential for the target figure. Unless otherwise specified, the bases are equilateral polygons, inscribed in the ellipse with major and minor semiaxes, designated as $\boldsymbol{B}_a$ and $\boldsymbol{B}_b$, or the ellipse itself. If the target has height in trivial geometric sense, it is always marked as $\boldsymbol{H}$. The geometric center

[(1)] my great gratitude to Mr.Leonid Olifer https://www.linkedin.com/in/leonid-olifer-43992b135/ for his efforts in testing of a new Target manager

of the base is "figure first point". The one of the base vertices always lies on the intersection of own X axis with the base ellipse. Number of vertices is $\boldsymbol{N}$. Circle or ellipse are based upon zero-vertices polygons. Perpendicular to the base built in figure first point is "figure main axis". In own coordinate triad every target is oriented along Z axis.

If target may be explained using the explanation of some another target, term `inherited` is used in the sense "the ball can be inherited from three-axial ellipsoid". Symbol $\rightarrow$ is used to show the inheritance.

Most of the targets are oriented. Orientation describes how the figure main axis (which is own Z axis) is placed, or in other words, which axis of the coordinate triad XYZ the figure main axis (always Z in own coordinates) is parallel to. Explanation of the elementary target is target name, followed with target parameters put in brackets. For example, the phrase `Flower(X, 7, 8, 9);` explains the target `Flower`, with main axis parallel to X axis of external coordinate frame and parameters: 7, 8, 9. Parameters are target-dependent.

It is possible to orient the elementary target along any of the axes of the external coordinate triad. It means just renaming of the axes. If main axis of the target is Y, it means that the target own Z axis is mapped on external Y, own Y on external X and own Y on external Z. These orientarion variants are processed by the code automatically.

The coordinate triads are always right oriented. Enumeration of the axes follows the circular rule. Example, if the target orientation is along Y axis, $\boldsymbol{B}_a$ should be the semiaxis along the next axis, Z in this case and $\boldsymbol{B}_b$ - along the X.

The full list of elementary targets with abbreviated list of parameters follows. Parameters are positional and will be extensively used in further explanations below. Additional explanations will be provided only for new abbreviations. Letter Q is used to show the placeholder for orientation axis. For the list of two-dimension or three-dimension points we use designation "2d list" and "3d list".

2.2.1 Orientable targets

$\boldsymbol{Pyramid}(\boldsymbol{Q}, \boldsymbol{N}, \boldsymbol{H}, \boldsymbol{B_a}, \boldsymbol{B_b})$ is a classic pyramid, main axis oriented along Q, base is N-vertices equilateral polygon, inscribed in the ellipse with semiaxes B_a and B_b.

$\boldsymbol{StarlikePyramid}(\boldsymbol{Q}, \boldsymbol{N}, \boldsymbol{H}, \boldsymbol{B_a}, \boldsymbol{B_b}, \boldsymbol{\mu}) \rightarrow \boldsymbol{Pyramid}(\boldsymbol{Q}, \boldsymbol{N}, \boldsymbol{H}, \boldsymbol{B_a}, \boldsymbol{B_b})$ is a pyramid built on starlike base. Here μ is the ratio of the base vertice central distances. For example, one can build starlike pyramid on the top of pentagram. In this case $\mu = BC/AC$, C - base center (see Fig. 4).

$\boldsymbol{Cone}(\boldsymbol{Q}, \boldsymbol{H}, \boldsymbol{B_a}, \boldsymbol{B_b}) \rightarrow \boldsymbol{Pyramid}(\boldsymbol{Q}, \mathbf{0}, \boldsymbol{H}, \boldsymbol{B_a}, \boldsymbol{B_b})$ is a pyramid with elliptical base (encoded as zero-vertices polygon).

$\boldsymbol{Tetrahedron}(\boldsymbol{Q}, \boldsymbol{a}) \rightarrow \boldsymbol{Pyramid}(\boldsymbol{Q}, \mathbf{3}, \boldsymbol{a}\sqrt{\mathbf{2/3}}, \boldsymbol{a}/\sqrt{\mathbf{3}}, \boldsymbol{a}/\sqrt{\mathbf{3}}), \boldsymbol{PlatoBody}$ is a classic Plato tetrahedron or a pyramid with equilateral triangle as a base with given edge length a.

Height and enscribing circle radius are calculated as $H = a\sqrt{2/3}$, $r = B_a = B_b = a/\sqrt{3}$.

$\boldsymbol{PolygonPyramid(Q, H, 2d\ list) \to Pyramid(Q, H, ...)}$ is a pyramid built atop of the polygon of general kind given as a list of 2d points. The base polygon may not be convex, but point (0,0,0) should lay inside. This is not tested and may cause inproper behaviour if ignored.

$\boldsymbol{Prism(Q, N, H, B_a, B_b)}$ is a classic prism.

$\boldsymbol{StarlikePrism(Q, N, H, B_a, B_b, \mu) \to Prism(Q, H, B_a, B_b)}$ is a prism built on starlike base like StarlikePyramid.

$\boldsymbol{Cylinder(Q, H, B_a, B_b) \to Prism(Q, 0, H, B_a, B_b)}$ is a prism built on elliptic base.

$\boldsymbol{Cube(Q, a) \to Prism(Q, 4, a, a/\sqrt{2}, a/\sqrt{2})}$, $\boldsymbol{PlatoBody}$ is a classic Plato cube and 4-vertices base prism build atop of square.

$\boldsymbol{PolygonPrism(Q, H, 2d\ list) \to Prism(Q, H, ...)}$ similar to PolygonPyramid.

$\boldsymbol{Parallelogram(Q, H, a, b, h_2) \to PolygonPrism(Q, H, ...)}$ is a prism based on a parallelogram as a base. See Fig. 2 for clarification. Figure first point is the parallelogram center.

$\boldsymbol{Trapezoid(Q, H, a, b, h_2) \to PolygonPrism(Q, H, ...)}$ is a prism based on a trapezoid as a base. See Fig. 3 for clarification. Figure first point is the trapezoid center, not the diagonal intersection point.

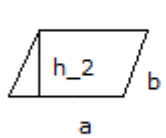


Figure 2: Parallelogram.

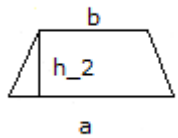


Figure 3: Trapezoid.

$\boldsymbol{Rectangle(Q, H, a, b) \to PolygonPrism(Q, H, ...)}$ is a prism based on a rectangle as a base. Figure first point is the rectangle center.

$\boldsymbol{Ellipsoid(Q, a, b, c, t)}$ is a classic tri-axial ellipsoid with the first point in its center. May be build filled with dipoles or only envelope with thickness t. Zero or negative thicknesses are ignored and filled figure is build in this case.

$\boldsymbol{Spring(Q, H, B_a, B_b, m, t) \to Cylinder(Q, H, B_a, B_b)}$ is a spring coiled counterclockwise from bottom on the cylinder, m - number of winds, t - tube radius (thickness).

$\boldsymbol{Cap(Q, a, b, c, t, h_2)}$ is a part of ellipsoid cut at distance h_2 from the top.

$\boldsymbol{Paraboloid(Q, H, B_a, B_b, t)}$ is a classic paraboloid.

$\boldsymbol{Brick(Q, a, b, c, t)}$ is a rectangular brick-like particle with coordinates started in its center.

$\boldsymbol{Box(Q, a, b, c, t)} \rightarrow \boldsymbol{Brick}$ is a rectangular box with an open top face.

$\boldsymbol{OpenBox(Q, a, b, c, t)} \rightarrow \boldsymbol{Brick}$ is a rectangular box with open top and bottom faces, or the tube of rectangular section.

$\boldsymbol{Torus(Q, R_a, R_b, r_a, r_b)}$ is an elliptical torus (semiaxes are R_a and R_b, R_a - perpendicular to Q, R_b - parallel to it) with elliptical cross section (semiaxes are r_a and r_b).

$\boldsymbol{TorusKnot(Q, R, r, \rho, p, q)}$ is a tube of radius ρ coiled around torus (main radius is R, second radius is r) p times in longitudinal and q times in meridional direction. Some example is presented in Fig. 5. If p and q are not relatively prime, then we have a torus knot with more than one component.

$\boldsymbol{Cassini(Q, a, e)}$ is a Cassini oval with focus coordinate a and eccentricity e.

$\boldsymbol{Polyhedron(Q, t, filename)}$ is a polyhedron of general kind in XYZ coordinates. Before using in **DDscat.C++** the polyhedron must be triangulated with any appopriate software, for example Qhull[(2)] or Tetgen[(3)]. We don't have any plans to include special triangulation codes in future releases of **DDscat.C++** . Each of the named codes is capable to create the input files for Polyhedron target.

First part of the file contains the vertice coordinates. These are (x, y, z) of the vertices, one vertex per line. This part of the file ends with the line containing only one asterisk symbol in the first position. The rest of the file are vertex indices of the triangles after triangulation, one triangle per line. After reading, the system check if there is zero index in the file, and if not, the system implies that the indices are given in fortran manner (starting from one, not zero). Example file `Polyhedron.inc` is included in this release. See it in `Misc` folder.

$\boldsymbol{Octahedron(Q, a), PlatoBody}$ are classic Plato bodies with the given edge size.

$\boldsymbol{Icosahedron(Q, a), PlatoBody}$

$\boldsymbol{Dodecahedron(Q, a), PlatoBody}$

$\boldsymbol{Lens(Q, R, r_1, r_2, \Delta)}$ is a lens-like target of radius R, limited by two spherical surfaces with radii r_1 and r_2. The radius should be given negative if curvature center is in negative direction of the surface. The distance between the surfaces measured along the lens optical axis is Δ.

2.2.2 Absolute targets

$\boldsymbol{Line(\rho, 3d\ list)}$ is a straight line (in the real life - the tube of radius ρ) of the segment(s) between all the specified points in given order. Any point is determined by their three coordinates. Number of points are not limited.

(2) http://www.qhull.org
(3) http://wias-berlin.de/software/tetgen/

$\boldsymbol{Plane}(\boldsymbol{\rho}, \vec{\boldsymbol{n}}, \vec{\boldsymbol{r}})$ is a plane with a given thickness ρ, passing through point $\vec{\boldsymbol{r}}$ normal to $\vec{\boldsymbol{n}}$.

$\boldsymbol{Grating}(\boldsymbol{\rho}, \vec{\boldsymbol{n}}, \vec{\boldsymbol{r}}, \boldsymbol{w_1}, \boldsymbol{w_2})$ is a plane-line grating with w_1 and w_2 are the widths of clean and filled strips respectfully.

$\boldsymbol{Arc}(\boldsymbol{\rho}, \vec{\boldsymbol{R}}_{\boldsymbol{1}}, \vec{\boldsymbol{R}}_{\boldsymbol{2}}, \vec{\boldsymbol{R}}_{\boldsymbol{3}})$ is a circular segment (in real life - the tube of radius ρ) passing through three specified points in given order.

$\boldsymbol{Circle}(\boldsymbol{\rho}, \vec{\boldsymbol{R}}_{\boldsymbol{1}}, \vec{\boldsymbol{R}}_{\boldsymbol{2}}, \vec{\boldsymbol{R}}_{\boldsymbol{3}})$ is a full circle (in real life - the tube of radius ρ) passing through three specified points in given order.

$\boldsymbol{Bezier}(\boldsymbol{\rho}, \boldsymbol{3d\ list})$ is a Bezier curve. It starts in the first point, ends in the last one and uses all intermediate points as a curve directions. Number of points are not limited. Elementary explanation can be found at, for example, Wikipedia. Some curve is given in Fig. 6.

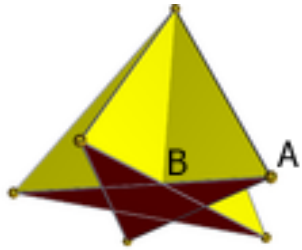


Figure 4: Starlike pyramid.

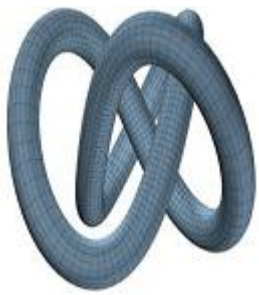


Figure 5: Torus knot.

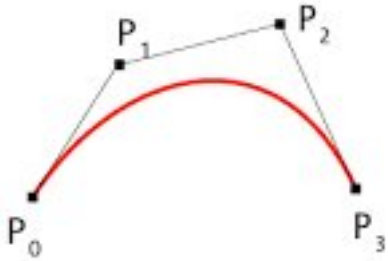


Figure 6: Bezier curve.

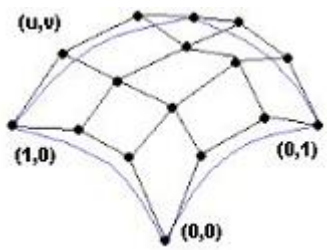


Figure 7: Bezier surface.

$\boldsymbol{Envelope}(\boldsymbol{\rho}, \boldsymbol{filename})$ is a Bezier surface built as a combination of 4×4 patches. Example of the patch is given in Fig. 7. To represent the envelope in general case one must use many patches. The vertex coordinates and mapping of the vertices into patches should be given in the file.

As in the case of Polyhedron the first part of the file contains the vertex coordinates, one vertex per line. This part of the file ends with the line with only one asterisk at the first position. The following lines contain the vertex indices of the patch, 16 indices (one patch) per line.

Graphics(***filename***) is a target created from the data stored in graphics file. The code of [**?**] was used as a source. (Only obj file processing is implemented.)

Gauss(***filename***) is a spherical particle created with the list of spherical garmonics. The copy of **DDSCAT** routine. (Can produce bugs and crashes.)

Fromfile(***filename***) is the target read from **DDSCAT** file. The copy of parent code routine.

Aggregate($\boldsymbol{\xi}$, ***filename***) is the target read from aggregate file. A lot of aggregates can be found on B.Draine personal page[(4)]. The only example BAM* file is in `Misc` folder. $\boldsymbol{\xi}$ is for multiplier, all dimensions read from aggregate file to be multiplied on.

2.2.3 Calculable target

There is the only target of this kind: ***Formula***($\boldsymbol{t}$, ***formula***). Formula is a mathematic expression to be used to construct the target. It is a receipt how to calculate the distance of the target surface point (with longitude λ and lattitude φ) to the target center.

The formula may contain numbers, binary plus (+), minus (-), multiply (*), divide (/), power (ˆ), and mathematical functions sin, cos, exp, log, sqrt, abs in their standard meaning (log is natural). Unary minus is not allowed. To represent longitude and lattitude (always in radians) user must use letters L and F. (If only F is used, the code will produce axially symmetric target). The target building system guarantees smooth surface. Mathematical π may be inserted in the formula with the letter P, Euler constant with an E. Any other symbols are not allowed and will cause the parsing error. Point at the end of formula is mandatory. Only square brackets are allowed to avoid the interference with C++ code ordinal parentheses. This is an example:

```
Formula(3, "40*abs[cos[4*F]*sin[5*L]].").
```

2.3 Target manipulation

User can manipulate the targets with the affine transform and logical operators. Target may be shifted and rotated. Let see it with an example:

```
Cone(Y, 20, 40, 30);
```

builds the cone, $H = 20$, base enscribing ellipse has semiaxes of 40 and 30. The cone main axis Z is collinear with external axis Y. It means that 40 is for cone semiaxis lying along the next external axis, namely Z (it is the next axis after Y) and 30 is for X semiaxis. The cone first point is at external point $(0, 0, 0)$. One may shift the cone:

(4) https://www.astro.princeton.edu/d̃raine/agglom.html

```
Cone(Y, 20, 40, 30) & ShiftA(10, -10, 20);
```

The cone just created is then shifted as a whole by $(10, -10, 20)$ vector. One may rotate it:

```
Cone(Y, 20, 40, 30) & ShiftA(10, -10, 20) & Rotate(X, 45);
```

This will turn the cone around its own X axis for 45° in positive direction (counterclockwise). There are two shift operators: `ShiftB` for shift vector given in laboratory coordinates, and `ShiftA` for shifts in own coordinates.

ShiftA, ShiftB and Rotate are operators to be applied to the figure with ampersand sign & in suffix position. In any case, ShiftA applies first, then Rotate and then ShiftB regardless of their order in target explanation. Multiply Shifts of the same kind for the same target are allowed but not recommended.

Targets may be manipulated by boolean logic operators. These are AND, OR and NOT (uppercase always). It is better to show it with an example again:

```
Cylinder(Y, 40, 30, 30) OR Cone(Y, 20, 30, 30) & ShiftA(0, 40, 0);
```

will create a complex target with all the dipoles of the cylinder OR the cone included. Before making OR, the target constructor code shifts the cone in Y direction by 40 points. As a result one will have the cone atop of the cylinder. And this joint target is oriented along external axis Y. To create the cones at both ends of the cylinder:

```
Cylinder(Y, 40, 30, 30) OR Cone(Y, 20, 30, 30) & ShiftA(0, 40, 0)
     OR Cone(-Y, 20, 30, 30);
```

The second cone is build upside-down and there are no shifts necessary. Yet another example: hexagonal prism with conical pits at both ends. This is:

```
Prism(Z, 6, 100, 20, 20) AND NOT Cone(Z, 40, 20, 20) & ShiftA(0, 0, 100)
     AND NOT Cone(-Z, 20, 40, 20);
```

It may look like hexagonal snow crystal from Fig 8 [(5)]. NOT operator here removes all the dipoles belonging to the cone specified. It creates the pits at the prism base ends.

Prisms and pyramids may be inclined. Inclination is introduced by two numbers in curly brackets after all parameters just before the closing parenthesis. In the following example the second cone is inclined. As Y is the main axis, the cone apex is shifted by 10 points in Z direction and 20 points in X direction, keeping the apex Y coordinate untouched.

[(5)] http://snowcrystals.com

```
Cone(Y, 20, 40, 30);
Cone(Y, 20, 40, 30, {10, 20});
```

The elementary targets may be cut. In the example below the cube is cut by the plane, holding throught point (15., 15., 30.) normal to vector (1., 1., 1.). Direction of normal defines which dipoles are cut off. If user imagine the normal vector like the arrow stuck in the cutting plane, the constructor removes the dipoles from the hemispace where head of the arrow is visible. Cut operator works in own coordinates. To remove the dipoles from another hemispace, change the signs in normal vector components:

```
Cube(Z, 30)&Cut(1., 1., 1., 15., 15., 30.);
```

The targets may me made anisotropic by applying the Compose operator:

```
Cube(Z, 30)&Compose(2, 3, 4);
```

As in parent code here 2, 3 and 4 are indices of the dielectric materials read from `*.par` file. These indices are started from one, not zero. If the target is isotropic, user may not apply Compose. In this case the target will be build with composition number one. Composition zero means vacuum. The system check is there is sufficient amount of dielectric material files mentioned in `*.par` file and generates the error if there is not.

The elementary target or their logical combinations may be cloned. There is an example:

```
Cone(Y, 20, 40, 30, {10, 20})^Clone(3, 20., 10., -10.);
```

here the inclined cone is copied three times along the direction of vector $(20, 10, -10)$. Any combination of the elementary targets put in brackets may be cloned too. Clone operator must be applied in suffix position with the help of hat (ˆ) sign. It is possible to clone already cloned target.

Periodic boundary conditions are introduced with global operator `Periodic`:

```
Cone(Y, 20, 40, 30, {10, 20})^Periodic(40., 30.);
```

Two parameters of Periodic are pyd and pzd in parent code designations. Clone and Periodic may be used together, but user should do that very attentively because the effect of Periodic on the generated target is not visible in ParaView.

2.4 Does the dipole belong the target

This section devotes to the only question: how to determine if the dipole at point $\vec{r}$ of given coordinates in XYZ frame belongs the constructed target? It may looks like some excerptions from computer geometry course but is it given here to be used as a reference.

For **all type of pyramids and prisms** we need a parameter, defined as p, see Fig 9. It is a distance from the point on a lateral surface with the same Z coordinate as the given point. It might be easily deduced as $p = -Z/H + 1$, and is equal $p = Z/H^2$ for Paraboloid. In a contrary to pyramids, $p = 1$ for all kinds of prism. Given point (x, y, z) belongs the target if the point (x, y) belongs the base polygon scaled p times.

For an **ellipse** the 2d point $p = (x, y)$ lies inside if: $(x/a)^2 + (y/b)^2 \leq 1$.

For an **ellipsoid** it is the same, but in 3d: $(x/a)^2 + (y/b)^2 + (z/c)^2 \leq 1$.

For a **torus:** hold a plane through the main axis of the torus and the given point. There are two ellipses of intesection. Put a point A in the center of the nearest ellipse. The given point, the point A and the torus center lie in that plane. The given point belongs the torus if it belongs the intersection ellipse.

For a **torus knot** its central line might be analytically represented as:

$$\vec{r} = ((R + r\cos p\psi)\cos q\psi,\ (R + r\cos p\psi)\sin q\psi,\ r\sin p\psi).$$

Hold the plane like it was done for torus. There are p circles in the intersection plane. Centers of those circles are equally spaced along the base torus circle. If α_i is the angle in the intersection plane from horizontal line to the circle center, it may be written as:

$$\alpha_i = \frac{q\psi p + 2\pi i}{q} = p\psi + \frac{2\pi i}{q}, \quad \text{with} \quad \tan q\psi = y/x, \quad i \in [0, q[.$$

A **Cassini** is 3d figure build by rotation of a 2d Cassini oval around the vertical axis. Its representation in polar coordinates is:

$$r^4 - 2a^2r^2\cos(2\theta) = b^4 - a^4, \qquad e = b/a,$$

where $(0, \pm a)$ are coordinates of the foci, and $e = b/a$ is an eccentricity. According to Wikipedia, if $e = 1$ Cassini oval is Bernoulli lemniscate, $e < 1$ - two circles (or deformed circles), $1 < e < \sqrt{2}$ - peanut like shape and $e \geq \sqrt{2}$ - convex ellipse-like figure.

Figure 8: Snow crystal.

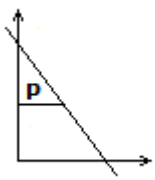


Figure 9: Definition of p.

For a **Spring** the intersection with vertical plane consists of circles n in number. The point belongs the spring if it belongs to some of the circles.

Point belongs a **Linear** tube between points $\vec{R}_1$ and $\vec{R}_2$ if $\Delta\vec{R} = \vec{R}_2 - \vec{R}_1$ and then

$$\alpha\Delta\vec{R} + \vec{x} = \vec{r} - \vec{R}_1, \qquad \alpha\Delta\vec{R}\cdot\vec{x} = 0,$$

$\vec{x}$ is perpendicular from given point onto $\Delta\vec{R}$, and $\alpha\Delta\vec{R}$ is a portion of $\Delta\vec{R}$ to the perpendicular base. Solving gives:

$$\alpha = \frac{\Delta\vec{R}(\vec{r} - \vec{R}_1)}{\Delta R^2}, \qquad 0 \le \alpha \le 1.$$

A distance from the given point to $\alpha\Delta\vec{R}$ should be less than ρ.

Center of an **Arc** or a **Circle** is determined by equation:

$$\vec{R} = \alpha_1\vec{r}_1 + \alpha_2\vec{r}_2 + \alpha_3\vec{r}_3,$$

where

$$\begin{aligned}
\alpha_1 &= \frac{a^2}{8S^2}(\vec{r}_1 - \vec{r}_2)\cdot(\vec{r}_1 - \vec{r}_3),\\
\alpha_2 &= \frac{b^2}{8S^2}(\vec{r}_2 - \vec{r}_1)\cdot(\vec{r}_2 - \vec{r}_3),\\
\alpha_3 &= \frac{c^2}{8S^2}(\vec{r}_3 - \vec{r}_1)\cdot(\vec{r}_3 - \vec{r}_2).
\end{aligned}$$

Here a, b, c - the sides of obvious triangle, $S = \sqrt{p(p-a)(p-b)(p-c)}$ - its square, $p = (a+b+c)/2$ - its halfperimeter. Its radius is

$$R = \frac{abc}{4S}.$$

Then continue in the Arc or Circle plane. Define three vectors lying in the plane ($\vec{r}$ is any point on the Arc or Circle):

$$\vec{\rho}_1 = \vec{r}_1 - \vec{R}, \qquad \vec{\rho}_3 = \vec{r}_3 - \vec{R}, \qquad \vec{\rho} = \vec{r} - \vec{R},$$

and two angles: ν - between $\vec{\rho}_1$ and $\vec{\rho}$, and μ - between $\vec{\rho}$ and $\vec{\rho}_3$. It can be shown that:

$$\vec{\rho} = \beta_1\vec{\rho}_1 + \beta_3\vec{\rho}_3,$$

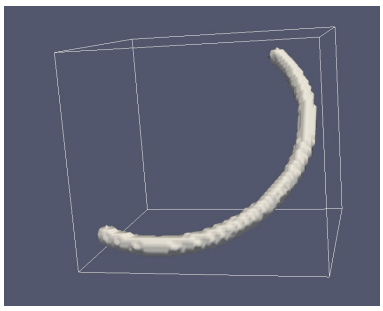
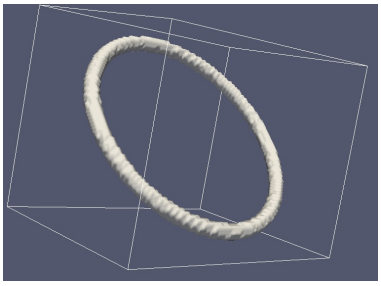
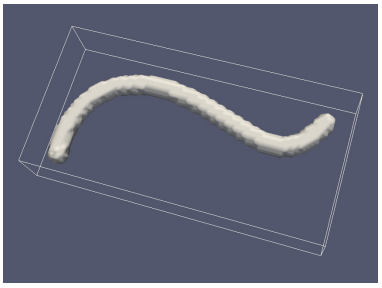
where

$$\beta_1 = \frac{\sin\nu}{\sin(\mu+\nu)}, \qquad \beta_3 = \frac{\sin\nu}{\sin(\mu+\nu)}.$$

For a **Bezier** curve, an **Envelope**, a **Plane** and a **Grating** we iterate through all candidate points, save them in the temporary set and then copy into a whole dipole list.

2.5 Gallery

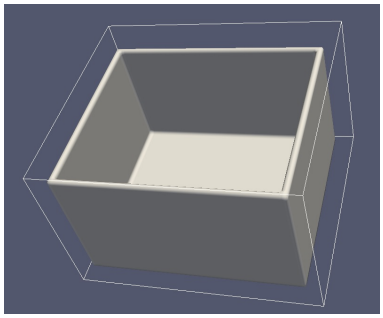
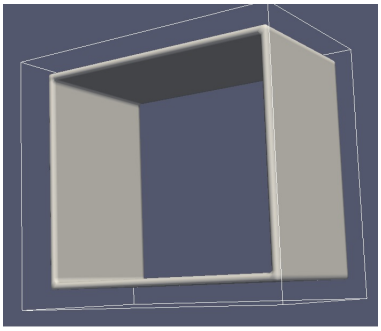
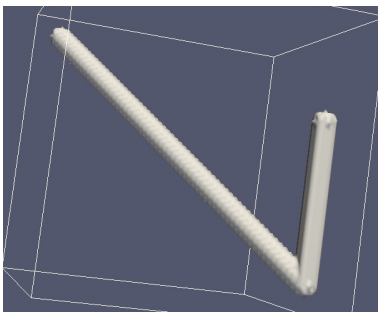
There is a gallery of the tagrets with the operators, used to build them. All input files for this gallery are stored in `Misc` folder.

Arc(3, -20., -20., -20., 40., 0., 0., 30., 30., 30.);

Circle(3, -20., -20., -20., 40., 0., 0., 30., 30., 30.);

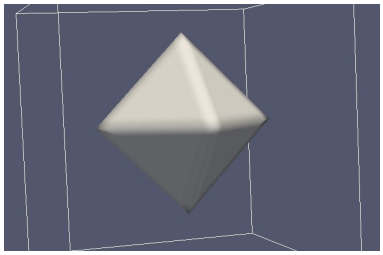
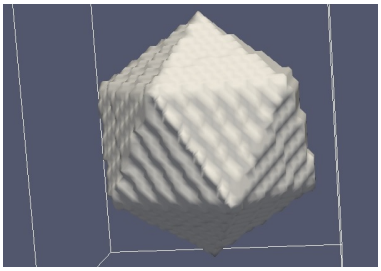
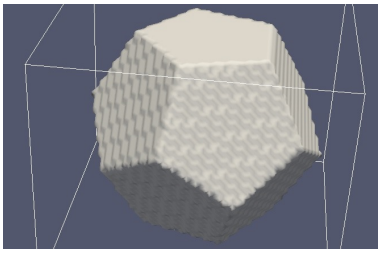
Bezier(3, -30., 0., 0., -35., 40., 0., 15., 40., 0., 30., 0., 0., 45., 30., 0.);

Box(Z, 70, 60, 40, 2);

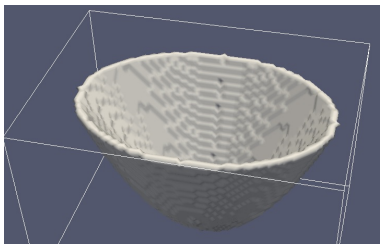
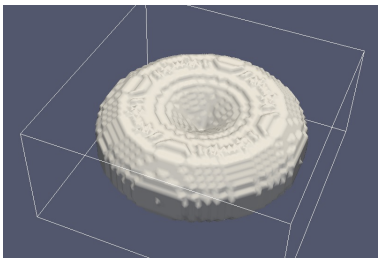
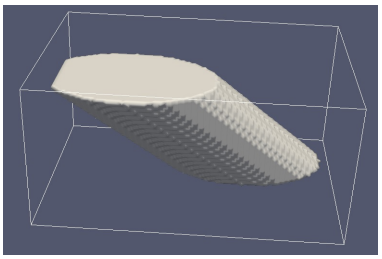
OpenBox(Z, 70, 60, 40, 2);

Line(6, -40., -40., -40., 20., 20., 20., -20., 20., 20.);

Octahedron(Z, 20); Icosahedron(Z, 20);

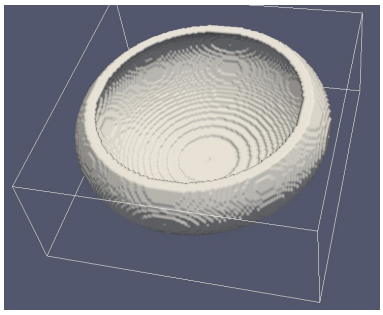
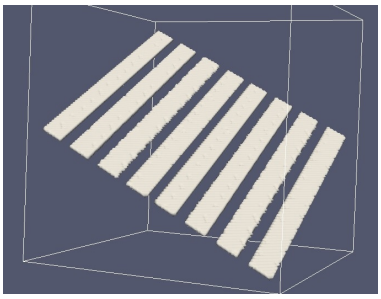
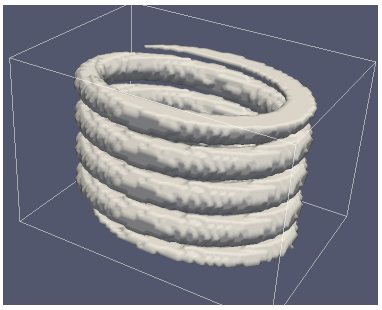
Dodecahedron(Z, 20);

Paraboloid(Z, 40, 30, 20, 4);

Cassini(Z, 40, 1.);

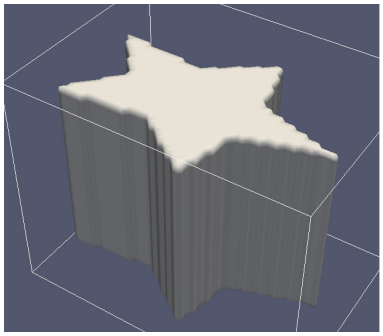
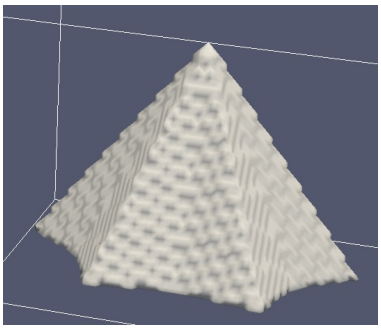
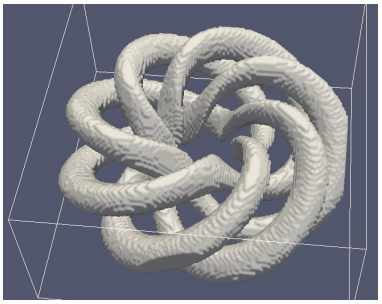
Cylinder(Z, 40, 30, 20, {40, 20});

Cap(Z, 60, 60, 30, 2, 45);

Grating(2, -1., -1., -2., 1., 5., -3., 5., 10.);

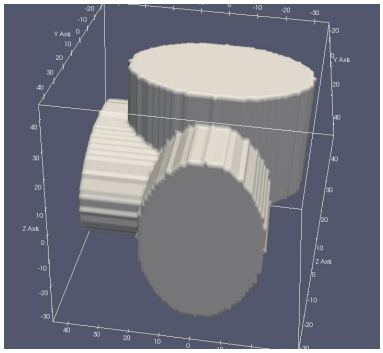
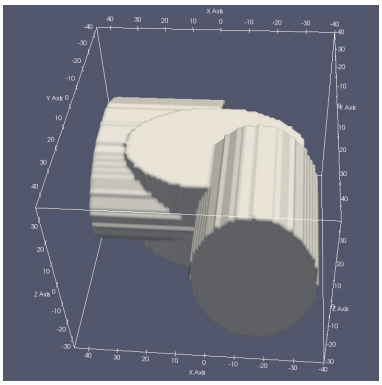
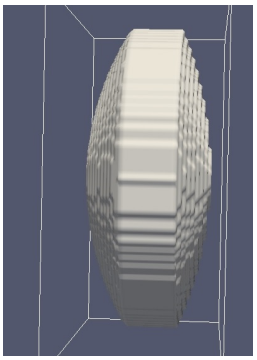
Spring(Z, 40, 30, 20, 4, 4);

StarlikePrism(Z, 5, 40, 30, 20, 0.5);

StarlikePyramid(Z, 6, 40, 30, 20, 0.75);

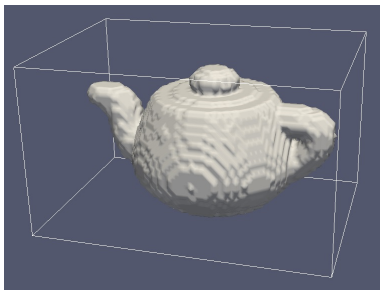
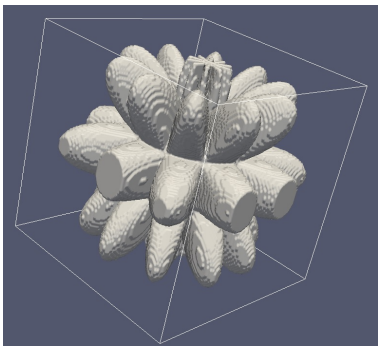
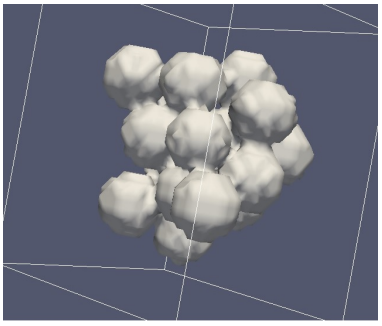
TorusKnot(Z, 40, 20, 3, 4, 7);

Cylinder(Z, 40, 30, 20) OR Cylinder(Y, 40, 30, 20) OR Cylinder(X, 40, 30, 20)

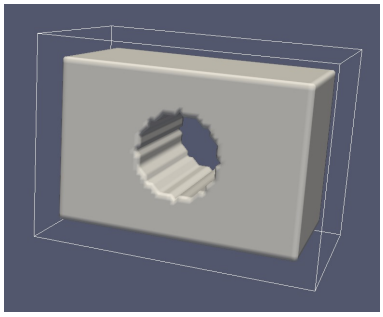
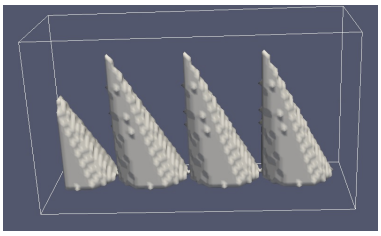
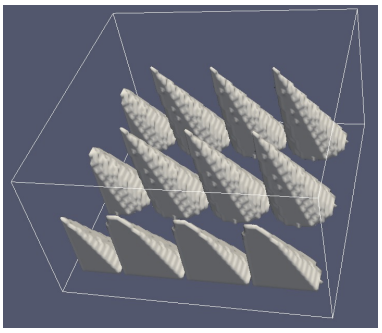
Cylinder(Z, 40, 30, 20)&ShiftA(0., 0., -20.) OR Cylinder(Y, 40, 30, 20)&ShiftA(0., -15., 0.) OR Cylinder(X, 40, 30, 20)&ShiftA(-10., 0., 0.);

Lens(Z, 40, 100, -75, 5.);

Envelope(3, ”../Misc/Envelope.inp”);

Formula(Z, 0, ”40+40*abs[cos[4*F]*sin[5*L]].”); FromFile(”../Misc/FromFile.inp”);

Brick(Z, 60, 40, 30, 0) AND NOT Cylinder(Z, 40, 10, 10)&ShiftA(0., 0., -20.);

Cone(Z, 40, 10, 10, {10, 10})ˆClone(3, 20., 0., 0.);

Cone(Z, 40, 10, 10, {10, 10})ˆClone(3, 20., 0., 0.)ˆClone(3, 0., 30., 0.);

2.6 BNF of target language

Add here some formal things. It is BNF grammar of the language for target explanation. The author is not absolutely confident that the grammar is Ok from formal side, but it is useful as a reference.

```
<target> := <figure>; | (<figure>); |<figure>^<global_operator>;
    | <figure> <op> <figure>;
<global_operator> := Clone(<clone_param>) | Periodic(<periodic_params>)
<clone_param> := <n>, <3d_point>
<periodic_params> := <3d_point> , <3d_point>
<op> := OR | AND | NOT
<figure> := <figure_movable> | <figure_absolute>
<figure_movable> := <figure_oriented> | <figure_oriented>&<operator>
<operator> := <opcode>(<param>) | <operator>&<opcode>(<param>)
<opcode> := ShiftA | ShiftB | Rotate | Compose | Cut
<param> := <value> | <param>, <value>
<figure_absolute> := <line> | <plane> | <arc> | <circle> | <bezier>
    | <gauss> | <figure_loadable>
<figure_loadable> := <graphics> | <fromfile> | <envelope> | <aggregate>
<figure_oriented> := <figure_inclinable> | <figure_formulable>
    | <3dsurfaces> | <bricks> | <polyhedra> | <torus> | <torusknot>
    | <lens>
<figure_inclinable> := <pyramids> | <prisms>
<figure_formulable> := <formula>
<3dsurfaces> := <ellipsoid> | <cap> | <paraboloid>
<bricks> := <brick> | <box> | <openbox>
<polyhedra> := <plato> | <polyhedron>
<pyramids> := <pyramid> | <starlikepyramid> | <cone> | <tetrahedron>
    | <polygonpyramid>
<prisms> := <prism> | <starlikeprism> | <cylinder> | <cube> | <spring>
    | <polygonprism> | <spring> | <misc>
<plato> := <octahedron> | <icosaedron> | <dodecahedron>
<misc> := <parallelogram> | <rectangle> | <trapeze>
<pyramid> := Pyramid(<ori>, <n>, <h>, <B_a>, <B_b>, <incliner>)
<starlikepyramid> := StarlikePyramid(<ori>, <n>, <h>, <B_a>, <B_b>, <incliner>)
<cone> := Cone(<ori>, <h>, <B_a>, <B_b>, <incliner>)
<tetrahedron> := Terahedron(<ori>, <a>, <incliner>)
<polygonpyramid> := PolygonPyramid(<ori>, <n>, <h>, <filename>)
    |  PolygonPyramid(<ori>, <n>, <h>, <2d list>)
<prism> := Prism(<ori>, <n>, <h>, <B_a>, <B_b>, <incliner>)
<starlikeprism> := StarlikePrism(<ori>, <n>, <h>, <B_a>, <B_b>, <incliner>)
```

```
<cylinder> := Cylinder(<ori>, <h>, <B_a>, <B_b>, <incliner>)
<spring> := Spring(<ori>, <h>, <B_a>, <B_b>, <winds>, <t>, <incliner>)
<cube> := Cube(<ori>, <h>, <incliner>)
<polygonprism> := PolygonPrism(<ori>, <n>, <h>, <filename>)
    | PolygonPrism(<ori>, <n>, <h>, <2d list>)
<ellipsoid> := Ellipsoid(<ori>, <semiaxis>, <semiaxis>, <semiaxis>, <t>)
<cap> := Cap(<ori>, <semiaxis>, <semiaxis>, <semiaxis>, <t>, <cut>)
<paraboloid> := Paraboloid(<semiaxis>, <semiaxis>, <semiaxis>, <t>, <cut>)
<brick> := Brick(<ori>, <a>, <b>, <c>, <t>)
<box> := Box(<ori>, <a>, <b>, <c>, <t>)
<openbox> := OpenBox(<ori>, <a>, <b>, <c>, <t>)
<octahedron> := Octahedron(<ori>, <a>)
<icosaedron> := Icosahedron(<ori>, <a>)
<dodecahedron> := Dodecahedron(<ori>, <a>)
<polyhedron> := Polyhedron(<ori>, <thickness>, <filename>)
<parallelogram> := Parallelogram(<ori>, <h>, <edge>, <edge>, <distance>)
<rectangle> := Rectangle(<ori>, <h>, <edge>, <edge>)
<trapezoid> := Trapezoid(<ori>, <h>, <base>, <base>, <distance>)
<lens> := Lens(<ori>, <R>, <r_1>, <r_2>, <t>)
<torus> := Torus(<R_a>, <R_b>, <r_a>, <r_b>)
<torusknot> := TorusKnot(<R>, <r>, <rho>, <p>, <q>)
<line> := Line(<t>, <3d list>)
<plane> := Plane(<t>, <normal>, <point>)
<grating> := Grating(<t>, <normal>, <point>, <empty>, <filled>)
<arc> := Arc(<t>, <3d list>)
<circle> := Circle(<t>, <3d list>)
<bezier> := Bezier(<t>, <3d list>)
<envelope> := Envelope(<t>, <filename>)
<graphics> := Graphics(<filename>)
<fromfile> := FromFile(<filename>)
<formula> := Formula(<ori>, <t>, <formula>)
<gauss> := Gauss(<filename>)
<aggregate> := Aggregate(<mult>, <filename>)
<incliner> := {<2d_point>} | NULL
<2d_list> := <2d point> | <2d point>, <2d list>
<3d_list> := <dd point> | <dd point>, <dd list>
<2d_point> := value, value
<3d_point> := value, value, value
<ori> := X | Y | Z | +X | +Y | +Z | -X | -Y | -Z
<h> := height
<r> := escribing radius
```

```
<t> := thickness
<mult> := multiplier
<filename> := string
```

3 Parameter file

Despite of the new target manager, the paremeter files left mostly unchanged.

Initial memory allocation line now must contain three or six numbers. First three numbers are upper limits of the rectangular area where the target is build. The second triplet is a lower limits. If they are omitted, the lower limits are set to zeros. It is possible to have lower limits negative.

Target geometry and composition lines are now the target headings and target receipt strings. This is an example:

```
'CYLINDER'        = CSHAPE*9 shape directive
'Cylinder(X, 64., 15., 15.);'
```

The receipt "line" may occupy as many lines as there is necessary to explain the target. Line break may be inserted anywhere but it is not allowed to break the words into parts. Strings for file names or formulae should be kept in the single line. Semicolon at the end of receipt string is mandatory. It is the point where the parser stops processing.

Just immediately after these lines, the lines with the compositions follow. The first of them is NCOMP line, the some amount of the following are the lines with file names in them. User is allowed to write 'water' or 'ice' as the file name, followed by the temperature in Kelvins. These two words force the **DDscat.C++** to use routines RefWat and RefIce from previous versions of **DDSCAT** . There are two files DataWater.txt and DataIce.txt in ../diel folder where the coefficients needed for these cases are stored. They are the part of the distribution.

Meaning of any other lines of the `*.par` file left unchanged.

The same changes must be inserted into xml par file. It is an example:

```
<Limits>
<MinVector X="0." Y="0." Z="0."/>
<MaxVector X="100." Y="100." Z="100."/>
</Limits>
<Target>
<Cshape Name="CYLINDER"/>
<Command Text="Cylinder(X, 64., 15., 15.);"/>
<Ncomp Amount="1">
```

```
<Dielec Pos="1" File="../diel/Au_evap"/>
</Ncomp>
</Target>
```

4 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[(6)]. These solvers are well explained in **DDSCAT** User Guide [1]. For **DDscat.C++** it is not mandatory to use CG codes. The user are allowed to add own solvers.

In version 7.3.1 of **DDscat.C++** the Solver engine had underwent a lot of cosmetic changes to be ready for further refactoring. But at this time the functionality left unchanged. It looks like we were able to resolve that QMRCCG solver bug in current release.

4.1 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")
```

4.2 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`;

(6) http://code.google.com/p/conjugate-gradient-lib/

- 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.

5 Field engines and managers

5.1 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 [9]. Usage of FFTW, Intel routines and CUDA versions are still disabled.

5.2 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.

5.3 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. At this time Dielectric manager can load and proceed with magnetic data files, but despite the fact that whole magnetic calculations are implemented, they still are not quite tested and are disabled.

6 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. Now it manipulates the log files too and creates the subfolder for any run of **DDscat.C++** in BinQt folder. That subfolder has the name based on the target name followed with D and date in YYMMDD format and then T and time in HHMMSS format.

If user want to have run log stored in log file, one must open log file by calling

```
OutputManager::GetInstance()->InitDirectory(currentTarget->Headings());
OutputManager::GetInstance()->OpenLogFile();
```

where Headings is the string used to name your target. It opens the log file `ddscat.xxx.log` where xxx is for user process number (zero for sequential calculations) in that just specified folder. Log file is managed by Output manager and is closed automatically at the end of **DDscat.C++** run. Wrimsg routine copy all its output into this file too. If user does not open log file, it will not be created but it will not influence the program run.

7 Postprocessing

In the current version (7.3.1) **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** 7.3.2, they are just subdivided into two parts. The par file for `Postprocess` is the same as for Readnf of **DDSCAT** 7.3.2.

8 Calltarget and DdaCovnert

Function of new CallTarget were explained earlier in this document. CallTarget2 is a new wxPython component to be released later and will help users to create the targets interactively.

DdaConvert is standalone code to transform graphics obj files into **DDscat.C++** readable format. Now DdaConvert uses `../Misc/test.obj` file and transfrom it into `../Misc/test.txt` one. That latter file can be loaded into **DDscat.C++** using FromFile target. Or, user can use Graphics target, giving the obj file name as its single parameter to load the obj file into **DDscat.C++** for processing. The only graphics file type implemented in obj. A lot of other file types are left for future releases.

9 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 author' personal page: `http://space.univ.kiev.ua/Choliy/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. This is the only possibility for the author to inform the engaged persons with news, bug fixes, new releases.

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, 66-70`

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. 2016, User Guide for the Discrete Dipole Approximation Code DDSCAT 7.3.2, `http://arxiv.org/abs/1305.6497v1.pdf`

[2] Choliy V.Ya. ...

[3] Choliy, V. 2013, "The discrete dipole approximation code DDscat.C++: features, limitations and plans". `Adv.Astron.Spa.Phys., 3, 66-70.`

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

[5] 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.`

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

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

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

[9] 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.`

[10] reference to ddscat mu code at ddscat site