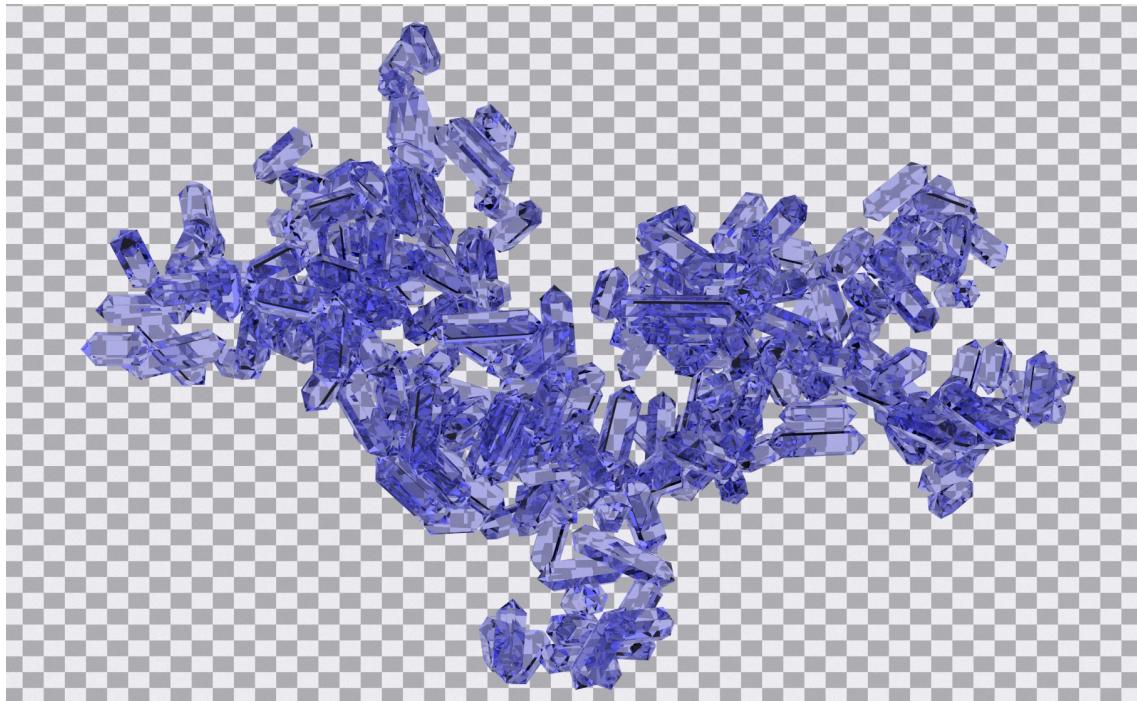


CHALMERS



A software toolkit for generating ice and snow particle shape data

Master's Thesis in COMPLEX ADAPTIVE SYSTEMS

Torbjörn Rathsman

Department of Earth and Space Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

A software toolkit for generating ice and snow particle shape data

MASTER'S THESIS

BY

Torbjörn Rathsmann

Department of Earth and Space Sciences
Chalmers University of Technology
Gothenburg, Sweden 2016

A software toolkit for generating ice and snow particle shape data
Torbjörn Rathsman

© Torbjörn Rathsman, 2016

RRYX03 - Master's thesis at Earth and Space Sciences
Supervisor: Patrick Eriksson
Examiner: Patrick Eriksson

Department of Earth and Space Science
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
+46 (31) 772 1000

Printed by Chalmers Reproservice
Göteborg, Sweden 2016

Cover: A picture of a snowflake generated by one of the algorithms implemented during this project. In order to show refraction effects simulated by the picture renderer, the snowflake has been placed in front of a checkerboard background.

Abstract

Ice and snow particle shape data are important for understanding the scattering of microwave radiation from a cloud. This thesis presents a software toolkit that can be used to generate such data, for use with Discrete Dipole Approximation calculations. The toolkit has been used to implement a Gillespie-based algorithm with overlap rejection. This algorithm, when used with hexagonal columns, has reproduced some of the properties of real snowflakes, namely their fractal dimension, and their size distribution.

The toolkit uses ice crystal prototypes to construct aggregates. Ice crystal prototypes can be modeled in 3D modelling software. This makes it is easy to construct exotic shapes, as opposed to a system based on different classes for different prototypes. In order to keep the possibility of arbitrary parameterisation, the ice crystal prototypes specifies transformation rules that are used if the ice crystal prototype should be deformed. Aggregates and ice crystal prototypes can be merged in different ways to form larger aggregates. To feed a DDA calculation program, the toolkit also provides a rasterisation system, which fills geometry with voxels by using a 6-directional floodfill algorithm.

A large part of the thesis discusses various ways of measuring particles. This has lead to a unit neutral way of testing whether or not a model simulates reality. The idea is to compare an averaged spherical volume fill ratio, which according to measurements should follow a particular equation, derived within this thesis.

Besides giving an overview of the toolkit, and presenting simulation results, this thesis also serves as a reference manual on how to use the toolkit.

Acknowledgements

This project would not have been without the idea from Patrick Eriksson. He has also suggested relevant quantities for the data analysis part of this project. A person who also should be mentioned is Robin Eklund for his feedback on the usability of the code, and his suggestions on improvements.

The author, April 2016

Contents

Notations used within this report	v
List of Figures	vi
List of Tables	viii
1 Introduction	1
1.1 About current research	2
1.2 The purpose of this project	3
1.3 Report outline	3
2 Construction of the toolkit	4
2.1 Ice particle synthesization	5
2.2 Storing ice particles	7
2.2.1 Intermediate aggregate storage	7
2.3 Mesh assembly	8
2.3.1 Deformation	9
2.3.2 Merging geometry	9
2.4 Geometry sampling	11
2.5 Utility routines	12
2.5.1 Face normal calculation	12
2.5.2 Aligning normal vectors	12
2.5.3 Testing whether or not a point is inside a domain	13
2.5.4 Centroid and volume calculation	13
2.5.5 Overlap detection	14
3 Models describing snowflake formation	17
3.1 A static “model”	17
3.2 A stochastic collision model (Model A)	17
3.2.1 Implementation in this toolkit	19
3.3 Non-spherical crystals and overlap detection	21
3.3.1 Non-spherical particles	21
3.3.2 Overlap detection	23
3.4 Adding more processes to the model (Model B)	23
3.4.1 Event probabilities	23
3.4.2 Storing probabilities	27

3.4.3	The resulting algorithm	28
4	Evaluating models	31
4.1	The relation between particle volume and radius	31
4.2	The particle size distribution	32
4.3	The spherical volume fill ratio	33
4.4	Data collection procedure	35
4.4.1	Collection of data from model B	35
4.5	Model parameter fitting	38
4.5.1	Dealing with time-dependent quantities	38
4.5.2	Finding quantity distribution among particles	42
4.5.3	Determining the fractal dimension	42
4.5.4	Determining particle size distribution	42
5	Results	43
5.1	The relation between particle volume and radius	43
5.2	The size distribution parameter	50
5.3	The spherical volume fill ratio	50
6	Discussion	58
6.1	Ways to improve the toolkit	58
6.2	Extensions and correctness of the models	59
6.3	Alternative ways of analyse data	59
6.4	Uncertainties in data	60
7	Conclusions	61
Glossary		62
Bibliography		64
A System requirements		66
B Retrieving and compiling the toolkit		67
C Predefined crystal files		69
D Command files used by the toolkit		70
D.1	General syntax	70
D.2	Ice crystal prototype definition files	71
D.2.1	Using Blender for creating ice crystal prototypes	73
D.3	Aggregate description files	74
D.3.1	Defining the graph	74

D.3.2	Declaring and retrieving parameters	77
D.3.3	Arithmetical transformation of parameters	77
D.3.4	Other commands	77
E	API reference	78
E.1	List of source files	78
E.2	Class reference	80
E.2.1	struct AggregateEdge	82
E.2.2	class AggregateGraph	83
E.2.3	class AggregateGraphLoader :public ConfigCommandHandler	83
E.2.4	class AggregateNode	84
E.2.5	struct BoundingBox	85
E.2.6	class ConfigCommandHandler	85
E.2.7	struct ConfigCommand	85
E.2.8	class ConfigParser	86
E.2.9	class ElementRandomizer	86
E.2.10	class FileIn	86
E.2.11	class FileOut	87
E.2.12	class IceParticle	87
E.2.13	class IceParticleVisitor	88
E.2.14	class MatrixStorage	89
E.2.15	class SolidBuilder :public IceParticleVisitor	90
E.2.16	class SolidBuilderBBC :public IceParticleVisitor	91
E.2.17	class Solid	91
E.2.18	class SolidDeformation	94
E.2.19	class SolidLoader :public ConfigCommandHandler	95
E.2.20	class SolidWriter	95
E.2.21	class Task	96
E.2.22	class Thread	96
E.2.23	class TicToc	96
E.2.24	struct Twins :public std::pair<T,T>	97
E.2.25	class VolumeConvex	97
E.2.26	class VolumeConvex :: Face	100
E.2.27	class VoxelBuilder	101
E.2.28	class VoxelBuilderAdda :public VoxelBuilder	102
E.3	Function reference	103
F	Result tables	104

Notations used within this report

Example	Description
<i>forward model</i>	A word explained in the glossary found at page 62
C	An arbitrary constant whose meaning depends on the context
O	The origin
\overline{AB}	A line segment between the points A and B
\mathbf{v}	A geometric vector
\overrightarrow{AB}	A geometric vector between the points A and B
$ \mathbf{v} $	The magnitude of a vector
$\hat{\mathbf{v}}$	A unit vector that is $ \hat{\mathbf{v}} = 1$
\mathbf{n}	A non-normalised face normal vector
$\hat{\mathbf{n}}$	A normalised face normal vector
\mathbf{A}	A matrix
\mathbf{I}	The identity matrix
\mathbf{X}	A general vector
k, l	Indices or integers
$\lfloor x \rfloor$	The integer part of x
$[x]$	The smallest integer grater than x
$\partial_x f$	The parial derivative of f with respect to x
$\nabla = \hat{\mathbf{x}}\partial_x + \hat{\mathbf{y}}\partial_y + \hat{\mathbf{z}}\partial_z$	The del operator
$\frac{d}{dx}$	The ordinary derivative with respect to x
\dot{x}	The time derivative of x
$\partial\Omega$	The boundary of Ω
$x \sim A$	The random variable x follows the probability distribution A
$\mathbb{E}(x) = \langle x \rangle$	The expectation value of x
$\text{std}(x)$	The standard deviation of x
$A \cap B$	The intersection between the sets A and B
\emptyset	The empty set
$\mathcal{O}(N)$	Algorithmic complexity. If $R(N)$ is the amount of resources used and the algorithmic complexity is $\mathcal{O}(N)$, then $\lim_{N \rightarrow \infty} \frac{R(N)}{N} = C$, where C is a constant.
Foobar	A class called Foobar used within a computer program
<code>foo</code>	An identifier in program source code
<code>foo</code>	A command or function called <code>foo</code>
foo	A programming keyword like nullptr
<code>Bar</code>	A program called <code>Bar</code>
<code>À</code> (U+0041)	Character with its Unicode codepoint

List of Figures

1.1	An illustration of the purpose of a forward model	1
1.2	Some common shapes used for simulating ice particle scattering	2
2.1	Possible data flows for generating snowflake shape data	6
2.2	A simplified class diagram showing the structure of the graph model of aggregates	8
2.3	An illustration of the mesh assembly process	10
2.4	An illustration of 6-directional flood fill	12
2.5	An illustration of the criterion used to test whether or not a voxel is inside a sub-volume	14
2.6	An illustration of the geometry used to derive expressions for testing triangle-triangle intersection	15
3.1	The data flow in model A	18
3.2	The merging scheme used in model A, without overlap detection	18
3.3	A comparison between an UV-sphere (left) and an icosphere (right) . .	20
3.4	An example of a “timber stack”	22
3.5	The data flow in model B	24
3.6	The layout of the probability matrix used by model B	27
4.1	A large and a small particle	33
4.2	An example of the dependency between the simulated time and the fractal dimension	39
4.3	The two possible shapes of a convergent time-series	41
5.1	The computed fractal dimension of particles generated by model B, as a function of different event rates	44
5.2	Correlation diagrams between the fractal dimension computed from two different runs of model B with different seed, but the same parameter sweep (different event rates)	45
5.3	The computed volume growth coefficient of particles generated by model B, as a function of different event rates	46
5.4	Correlation diagrams between the volume growth coefficient computed from two different runs of model B with different seed, but the same parameter sweep (different event rates)	47

5.5	Relation between crystal prototype length and the fractal dimension for particles generated by model B	47
5.6	Correlation diagrams between the fractal dimension computed from two different runs of model B with different seed, but the same parameter sweep (different geometry)	48
5.7	Relation between crystal prototype length and the normalised volume growth coefficient for particles generated by model B	48
5.8	Correlation diagrams between the volume growth coefficient computed from two different runs of model B with different seed, but the same parameter sweep (different geometry)	49
5.9	Particle volume as a function of their size for particles generated by model B	49
5.10	The size distribution parameter among particles generated by model B, as a function of different event rates	51
5.11	Correlation diagrams between the fractal dimension computed from two different runs of B with the same parameter sweep (different event rates)	52
5.12	Relation between crystal prototype length and the size distribution parameter among particles generated by model B	52
5.13	Correlation diagrams between the fractal dimension computed from two different runs of B with the same parameter sweep (different geometry)	53
5.14	Size distribution among particles generated by model B	53
5.15	Size distribution among dropped particles generated by model B	54
5.16	The pseudo-average spherical volume fill ratio of particles generated by model B, as a function of different event rates	55
5.17	Correlation diagrams between the pseudo-average spherical volume fill ratio computed from two different runs of model B with different seed, but the same parameter sweep (different event rates)	56
5.18	Relation between crystal prototype length and the pseudo-average spherical volume fill ratio of particles generated by model B	56
5.19	Correlation diagrams between the average spherical volume fill ratio computed from two different runs of model B with different seed, but the same parameter sweep (different geometry)	57
5.20	The reduced spherical volume fill ratio as a function of particle size	57

List of Tables

2.1	Some predefined crystal shapes bundled with the toolkit	5
4.1	The quantities recorded during simulations	36
4.2	Parameter values used to evaluate model B	37
4.3	The parameter values used to evaluate model B with respect to different geometry	37
C.1	Crystal prototypes bundled with the toolkit.	69
D.1	Delimiter characters and their function during the parsing process of command files	70
D.2	Commands allowed in an ice crystal prototype definition file	71
D.3	Commands allowed in an aggregate description file	76
F.1	Computed values of the fractal dimension of particles generated by model B for different event rates	104
F.2	Computed values of the normalised volume growth coefficient of particles generated by model B for different event rates	105
F.3	Computed values of the fractal dimension of particles generated by model B with different crystal prototype length	106
F.4	Computed values of the volume growth coefficient of particles generated by model B with different crystal prototype length	106
F.5	Computed values of the size distribution parameter among particles generated by model B	107
F.6	Computed values of the size distribution parameter among particles generated by model B with different crystal prototype length	108
F.7	Computed values of the mean spherical volume fill ratio of particles generated by model B	109
F.8	Computed values of the mean spherical volume fill ratio of particles generated by model B	110

1 Introduction

Both weather forecasting and climate model verification are today dependent on satellite observations. Many such observations are dependent on a reliable *forward model* (see fig. 1.1), since the quantities of interest cannot be measured directly. By inverting the forward model, it is possible to retrieve the quantities of interest. This requires that all parameters of the forward model can be found. Finding parameters can be done by measuring pairs of \mathbf{Y} and \mathbf{X} , and then fitting the model to all data points. By using the fitted parameters, it should then be possible to get \mathbf{X} from \mathbf{Y} or vice versa. It may also happen that the parameters themselves are of main interest. In the context of weather and climate modelling, one such parameter is the integrated water content of a cloud.

The integrated water content of a cloud is related to the amount snow and ice particles in the cloud. An important effect of such particles is the scattering from them. Understanding the scattering is needed for getting a correct data retrieval, since it has a great impact on the measurement. For example, if it is of interest to measure the temperature of a radiating object below an ice cloud, the scattering from the cloud will affect the measurement. The scattering from an object, in this case the particles inside the cloud, is among other things dependent of its shape, and therefore, there is a need for facilities that can generate realistic particle shape data.

It is in general impossible to find analytical expressions for the scattering, and therefore computer simulations are needed in order to find out how different particle mixtures affects scattering. Of particular interest is microwave scattering, since the frequencies used for measurement is in the range 50 GHz to 800 GHz, which corresponds to wavelengths of size 0.4 mm to 6 mm. This is approximately the same size as the particles of interest and therefore, the scattering process is called *Mie scattering* (Rees, W. G. 2001, p72-73). Simulations within this wavelength region

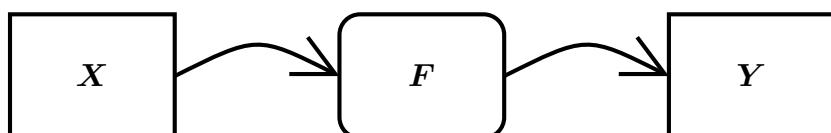


Figure 1.1: An illustration of the purpose of a forward model. The quantity \mathbf{Y} is measured, and the interesting quantity is \mathbf{X} . By inverting the forward model \mathbf{F} , it is possible to find \mathbf{X} from measured data.



Figure 1.2: Some common shapes used for simulating ice particle scattering

can be performed using the *Discrete Dipole Approximation* (DDA)¹ method (Draine, B. T. et al. 1994). The DDA technique requires the particle volume to be filled with a three-dimensional raster of electric dipoles (Draine, B. T. et al. 1994). To be able to do this, the particle shape has to be known.

1.1 About current research

Current research has shown that the mass-radius relation for ice particles in cirrus clouds is a power law, that is $m = CR_{\max}^{\beta}$, with $\beta \approx 2$ (Baran, A. J. 2012). The size distribution of the ice particles is somewhat more uncertain, especially the amount of smaller particles (those of size less than 250 μm), due to the destructive measurement techniques used (Baran, A. J. 2012). Nevertheless there are measurements of ice particles that indicate that their size distribution is exponential (Garrett, T. J. et al. 2015), and the range of measured particle sizes spans 10 μm to 1000 μm and larger (Baran, A. J. 2012).

There exists a number of models describing the shapes of ice particles. Many of these models do not include any dynamics. Rather, they are based on static distributions of the particle shapes and sizes (Baran, A. J. 2012), like those shown in fig. 1.2. There are also some models which involve dynamics. One such model, which is described by Maruyama, K. et al. (2005), is a stochastic model driven by coalescence probabilities of all particles inside a cloud. While the authors chose to include only spherical particles for the initial set of particles, and did not consider event types other than particle coalescence, the model can in principle be extended to cover any initial mixture of particle geometry, and it is also possible to add other event types such as new particle creation, particle melting, and particle drop-out.

¹A glossary is found at page 62

1.2 The purpose of this project

In order to understand more about the scattering from ice particles, there is a need for software that can, given a more flexible geometry representation, generate three-dimensional raster data suitable for DDA calculations, and that can be used to simulate the formation of ice particles. With a more flexible geometry representation, it is possible to build complex geometries out of simple primitives, with an accuracy only limited by machine precision. This would not be possible with the DDA-friendly raster, unless it has an impractical amount of raster points. The possibility of building complex geometries for use with DDA calculations makes it easier to simulate how different particle shapes affect microwave scattering. Particle formation can also be thought of as building complex structures out of primitives by following certain rules. Therefore, the ability to assemble geometry from primitives can also be used when simulating particle formation.

The goals of this project have been to implement code that can generate input data to be used in DDA calculations, and to implement at least one model that simulates ice or snow particle formation. The first task has been done by first creating a set of basic crystal shapes shapes, that can be imported through an aggregate description file, and then rasterised. For the second task, the basic crystal shapes has been used as input for the implemented models.

For the modelling part, the main focus has been on extending the model suggested by Maruyama, K. et al. (2005), and to some extent, analyse its output. The extensions made to the model are to include the possibility of different initial shapes other than spheres, as well as to include new particle formation, particle melting, and particle drop-out.

1.3 Report outline

Since the project has focused on implementing “A software toolkit for generating ice and snow particle shape data”, the main part of the report begins with a description of how different parts of the toolkit are implemented. In the following chapter, models for ice particle formation are discussed. Chapter 4 is dedicated to model evaluation through analysis of data collected from the raw output of simulations. Some results of the results from the analysis are presented in Chapter 5. The results are discussed further in Chapter 6, which also contains suggestions on how the work done during this project can proceed.

2 Construction of the toolkit

Rather than building a monolithic application, the toolkit works as a software library. The main reason for that is that it should be possible to use different algorithms to generate snowflakes. By implementing a common set of subroutines, algorithms can use those subroutines in order to generate snowflakes. This chapter is dedicated to explain the theory behind these subroutines. A programmer's view on the *Application Programming Interface* (API) can be found in appendix E. Most of the subroutines implement algorithms from computational geometry, making the toolkit more or less a geometry engine. There are also functions for loading and storing geometry in different file formats.

Despite a general approach, some assumptions has to be made. As a starting point, it is assumed that a program based on the toolkit requires some parameters, and a set of basic ice crystal geometries, that can be used to create more complex shapes. The basic crystal geometries are hereafter called *crystal prototypes*. Some common shapes bundled with the toolkit are shown in table 2.1. Instead of using the bundled shapes, the user can also draw custom shapes by using 3D modelling software, or write them manually by using a text editor.

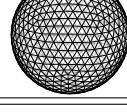
The toolkit does not assume any particular set of measurement units. Since the toolkit is more of a geometry engine than a physics engine, the only unit of interest is the unit of length, which is assumed to be the same as the one used when drawing the crystal prototypes. This unit is called *system unit*. If the user does not convert the output quantities before storing data, this unit will also be used when exporting data.

The toolkit is constructed around the data flow outlined in fig. 2.1. It consists of four major stages: **Particle synthesization**, **Mesh generation**, **Geometry sampling**, and **Data collection**. The input data mentioned above are collectively named “Static resources” in fig. 2.1. Although this picture has been used for inspiration when implementing the toolkit, the toolkit does not force any execution order. In other words, the toolkit does not use *inversion of control* to force a particular data flow, anywhere but in a few places where data traversing seemed to be complicated. An overview of the source code can be found in appendix E.

In the first stage, ice particles are assembled using some model. The model can do anything from assembling prescribed shapes, to perform a more complete physical simulation, using other already computed results as input.

The ice particle synthesization stage can generate a *polygon mesh* directly as well as using a *graph* structure as an intermediate form. If the former approach is used,

Table 2.1: Some predefined crystal shapes bundled with the toolkit. A complete list of crystal prototypes together with their parameters can be found in table C.1

Shape	Name	Description
	bullet	A single-ended hexagonal bullet. The tip height can be reduced to zero, which turns the bullet into a column
	hollow	A hollow column. The hole depth can be reduced to zero, which turns the hollow column into a solid column
	spheroid	A spheroid. The spheroid can be stretched along any orthogonal set of axes.

the synthesization algorithm can determine to merge ice particles and then, they form a new ice particle directly. If the latter approach is used, the mesh assembly stage converts the graph structure into polygon meshes.

The final polygon mesh can be fed into the geometry sampler to generate DDA input data. The geometry sampler fills the generated ice particles with virtual electric dipoles. To create measurements of the ice particles produced, there are also various data collection procedures, that measure geometric quantities of an ice particle. Since the mesh follows a conventional structure, it is possible also to export the mesh and import it into other tools.

2.1 Ice particle synthesization

The particle synthesization process is responsible for creating aggregates using some physical model, unknown to the toolkit. Therefore, what happens in this stage is fully defined by the chosen physical model. The models implemented during this project are outlined in chapter 3. The responsibility of the toolkit is to provide the basic functionality needed to implement the physical model. All models implemented in this project are based on physically motivated “copy-paste” processes that need routines for processing geometry through deformation, rotation, placing, and merging.

As described above, the toolkit support two different ways of assembling an ice particle: a **direct** method without intermediate storage, and an **indirect** method with intermediate storage. In common, both methods formalise the way of merging

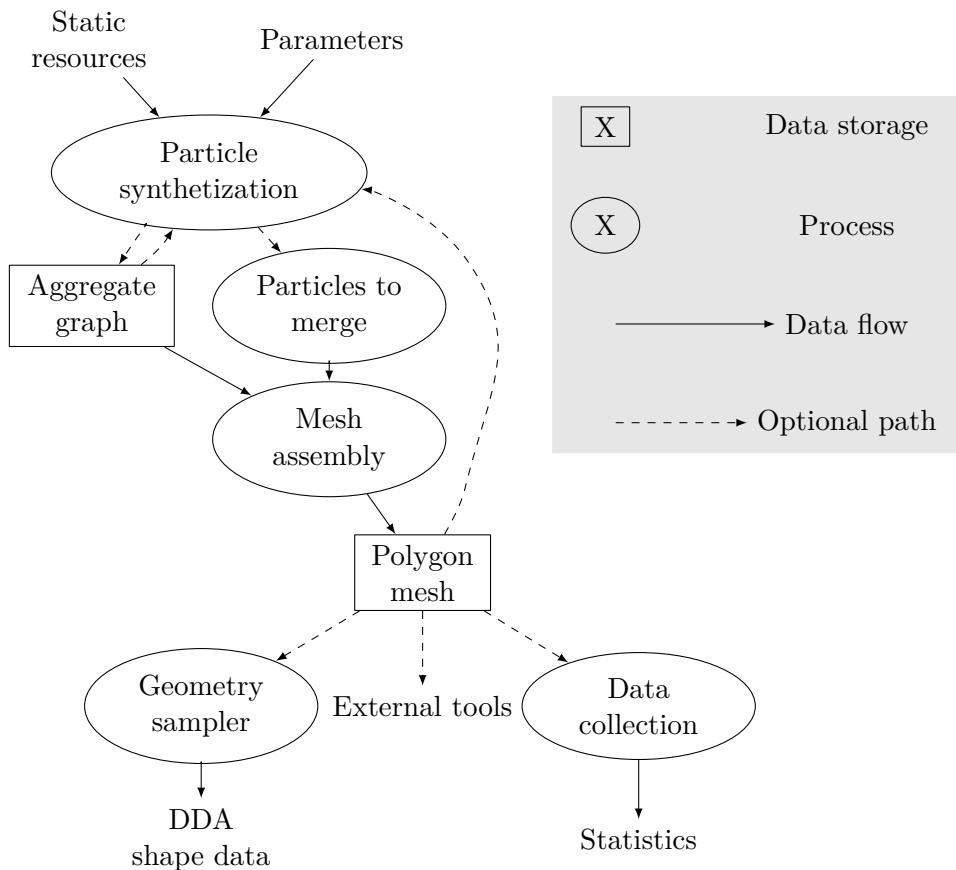


Figure 2.1: Possible data flows for generating snowflake shape data. The particle synthesization stage is responsible for the assembly of the geometry, while the mesh generator constructs a polygon mesh. The particle synthesization stage can both create a polygon mesh directly and use the intermediate aggregate graph storage for later assembly. The polygon mesh is used to generate DDA input data, or polygon mesh statistics. The polygon mesh can also be exported for other usage.

two polygon meshes. With the direct method, these polygon meshes are directly merged through a transformation matrix specified by the caller. As mentioned above, the indirect method uses a graph structure which connects polygon meshes with offset vectors and bond angles. The indirect method can be used for models that are more easily formulated from angles and distances than using absolute positions, or models that need to break an ice particle while preserving its parts. If the model does not need these features, it should be implemented using the direct method instead, since it avoids the possibly large overhead in converting the graph to a polygon mesh.

2.2 Storing ice particles

The most fundamental building block used for storing ice particles is a *sub-volume*. A sub-volume consists of a non-loose polygon mesh described as vertices and faces, each face referring to three vertices. An important feature of a sub-volume is that it is a *convex set*. To emphasise this assumption, they are called **VolumeConvex** within the toolkit. Besides vertices and faces, a sub-volume also maintains a list of faces that are assumed to be visible, and associations between vertex groups and vertices.

To be able to construct non-convex geometry, there are **Solids**. A **Solid** is a collection of sub-volumes and a set of transformation rules, each described by a 4×4 matrix with parameterised entries. The transformation rule also has a name, which associates it with a vertex group.

The next level of abstraction is an **IceParticle**. An **IceParticle** adds non-geometrical information such as velocity and density. It also keeps its own set of parameters for the geometry transformations. In this sense, a **Solid** works as a template when forming an **IceParticle**.

2.2.1 Intermediate aggregate storage

When the indirect approach for generating an ice particle is used, the structure of the new particle is stored as a graph illustrated in fig. 2.2. Each node has an **IceParticle**, and contains all *edges* that connects the **IceParticle** of the current node to all its neighbours. In addition to storing pointers to the two nodes that are connected through the edge, the edge also stores geometric data used for the mesh generation process illustrated in fig. 2.3. The geometric data consists of

- the offset vector \mathbf{u} from the centre of the parent **IceParticle** P_0 to the joint anchor point A in coordinates of the parent **IceParticle**
- the offset vector \mathbf{v} from the centre of the child **IceParticle** P_1 to the joint anchor point B in coordinates of the child **IceParticle**

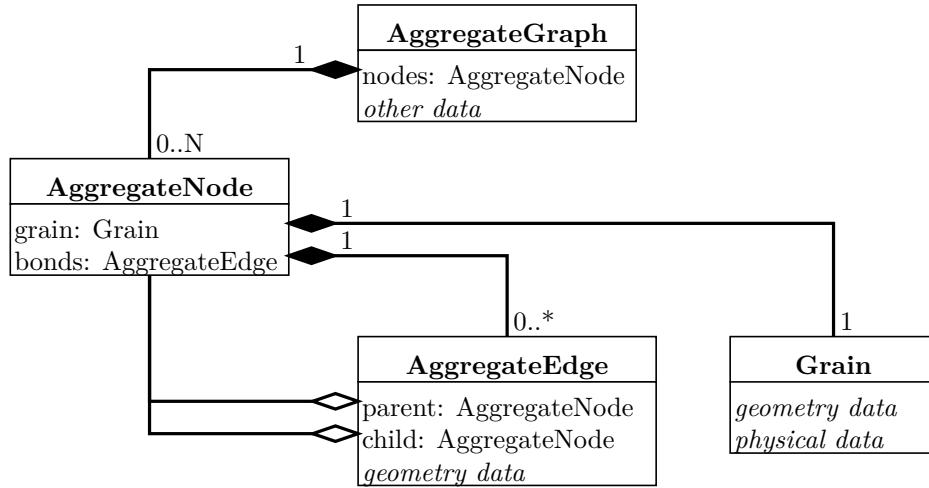


Figure 2.2: A simplified class diagram showing the structure of the graph model of aggregates. A solid diamond at class *A* marks that the data described by the class at the other end of the association are stored together with the data described by *A* (An **IceParticle** is stored together with an **AggregateNode**), and a hollow diamond marks that the data is stored somewhere else (An **AggregateEdge** contains two **AggregateNode** pointers).

- the turn angles—illustrated by $\Delta\theta$ —to rotate by, when moving from the parent node to the child node.

In order to keep the mesh generation process independent of the choice of starting node, both the parent and child node keep an own version of the binding edge (the class **AggregateEdge** in fig. 2.2). In the child version, the two node pointers, as well as the vectors \mathbf{u} and \mathbf{v} are swapped, and the turn angles are ordered backwards and have opposite sign.

2.3 Mesh assembly

The mesh assembly process is responsible for creating composite **IceParticles** from other **IceParticles**. The process takes two already existing **IceParticles** *A* and *B* and creates a new **IceParticle** composed of *A* and *B*. The process can be described as a two-stage process, an optional deformation step followed by the merging of the particle geometries.

2.3.1 Deformation

Before merging **IceParticles**, any newly added **IceParticles** are deformed by applying the transformation rules (see section 2.2) associated with the underlying **Solid**. This step makes it possible to do continuous deformations of the corresponding crystal prototype, without having to store one crystal prototype per imaginable mesh deformation.

When an **IceParticle** is deformed, vertices are moved through the matrix transformation specified by the corresponding transformation rule. The 4×4 matrix of the transformation rule is an ordinary geometry transformation matrix, and this matrix is applied to all vertices within the group with the same name as the transformation rule. As an example of how the deformation system works, consider the transformation rule `foo`. If the transformation has the matrix

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & t \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

all vertices within the group `foo` will be moved t units to the right. The parameter t may be specified any time during runtime.

No deformation must break the assumption that each sub-volume is a convex set. There is also a reserved group called `$global`, which affects all vertices within the **IceParticle**.

2.3.2 Merging geometry

To locate and orient the **IceParticles** to each other, the offset vectors \mathbf{u} and \mathbf{v} (see fig. 2.3) are used, together with the turn angles θ and $\Delta\theta$. More precisely, the **IceParticles** are placed by locating B at A , and by rotating the **IceParticles**. Given the designations in fig. 2.3, it holds that

$$\begin{cases} \mathbf{r}_a = \mathbf{r}_o + \mathbf{u} \\ \mathbf{r}_b = \mathbf{r}_1 + \mathbf{v} \end{cases}$$

To find a relation between the position vector \mathbf{r}_o and the position vector \mathbf{r}_1 , the constraint $A = B \implies \mathbf{r}_a = \mathbf{r}_b$ is used. This gives

$$\mathbf{r}_o + \mathbf{u} = \mathbf{r}_1 + \mathbf{v} \Leftrightarrow \mathbf{r}_1 = \mathbf{r}_o + \mathbf{u} - \mathbf{v}$$

Since \mathbf{u} and \mathbf{v} are expressed in the coordinates of the corresponding **IceParticle**, both \mathbf{u} and \mathbf{v} have to be rotated. In the two-dimensional case drawn in fig. 2.3, the correct expression for \mathbf{r}_1 becomes

$$\mathbf{r}_1 = \mathbf{r}_o + \mathbf{R}(\theta)\mathbf{u} - \mathbf{R}(\theta + \Delta\theta)\mathbf{v} = \mathbf{r}_o + \mathbf{R}(\theta)(\mathbf{u} - \mathbf{R}(\Delta\theta)\mathbf{v})$$

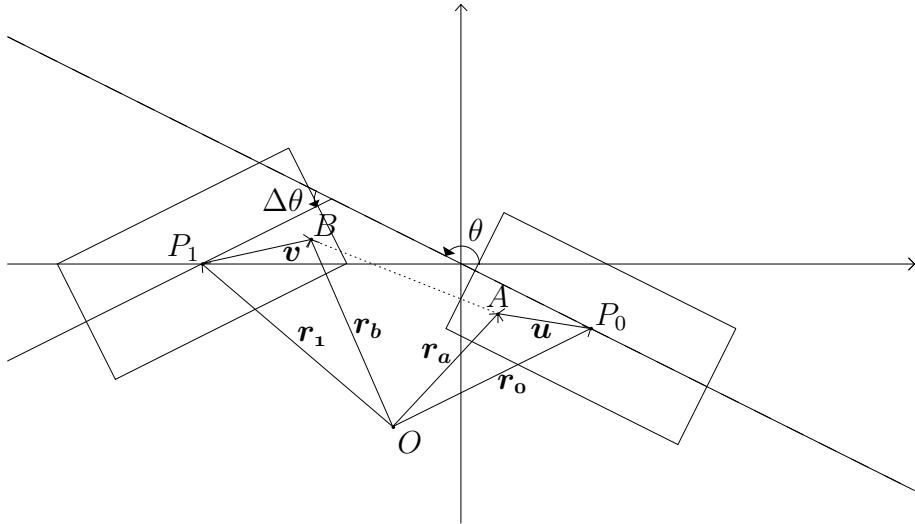


Figure 2.3: An illustration of the mesh assembly process. Two **IceParticles** are being merged at the anchor points $A = B$. The points are identified by the vectors \mathbf{u} and \mathbf{v} respectively. To make the figure more clear, only the two-dimensional case is drawn. The code works in 3D.

where \mathbf{R} is the rotation matrix. In 3D, the rotation matrix is composed by three matrices—one for each rotation axis, which gives

$$\mathbf{r}_1 = \mathbf{r}_o + \mathbf{R}_z(\theta_z)\mathbf{R}_y(\theta_y)\mathbf{R}_x(\theta_x)(\mathbf{u} - \mathbf{R}_z(\Delta\theta_z)\mathbf{R}_y(\Delta\theta_y)\mathbf{R}_x(\Delta\theta_x)\mathbf{v})$$

When the direct approach is used, the rotation matrix has to be specified rather than the turn angles. To make it easier to construct such a matrix, there is a function, described in section 2.5.2, that constructs a rotation matrix given two normal vectors.

When the indirect approach is used, the turn angles are stored in the graph edges described in section 2.2.1. To assemble the full ice particle, the graph is traversed using the *depth first* approach. After two **IceParticles** have been merged, the current coordinate system state (its origin and orientation) case updated according to the rules

$$\begin{cases} \mathbf{r}_1 \rightarrow \mathbf{r}_o \\ \theta_x + \Delta\theta_x \rightarrow \theta_x \\ \theta_y + \Delta\theta_y \rightarrow \theta_y \\ \theta_z + \Delta\theta_z \rightarrow \theta_z \end{cases}$$

To be able to recover the old state when a branch is finished, the coordinate system state is stored on a *stack* before processing the nodes of a branch. When all nodes in the branch has been processed, the old coordinate system state is restored from the stack.

The graph traversal is done by first pushing the next child of the current node onto a stack, and then by pushing the first grandchild onto the stack. The latter is only pushed if it has not yet been visited. Nodes are popped subsequently as long as there are nodes left on the stack.

2.4 Geometry sampling

The geometry is sampled by using a 6-directional flood fill algorithm—illustrated in fig. 2.4—on each sub-volume in the ice particle. This way, the entire ice particle is filled, and each location is only filled once. The flood fill algorithm starts at the current sub-volume’s geometric mass centre C , which is guaranteed to be inside the sub-volume due to the assumption of it being a convex set.

The flood fill algorithm requires discretisation of the ice particle volume. The discretisation is done by dividing the axis-aligned *bounding box* of the assembled ice particle into $N_x \times N_y \times N_z$ elements or *voxels*. The coordinate quantisation that goes from a continuous coordinate x to a discrete coordinate k is

$$k = \left\lfloor \frac{N_x(x - x_{\min})}{x_{\max} - x_{\min}} \right\rfloor \quad (2.1)$$

The “inverse” formula is

$$x = x_{\min} + \frac{(k + \frac{1}{2})(x_{\max} - x_{\min})}{N_x} \quad (2.2)$$

where $k \in [0, N_x \cap \mathbb{Z}]$. The initial value of k is given by eq. (2.1) evaluated at $x = C_x$. During the execution of the algorithm, its value is changed by ± 1 as long as the value stays inside the allowed boundaries, the x given by eq. (2.2) is inside the sub-volume, and the voxel at the next k is not yet visited. Similar expressions are also used for the y and z coordinates.

In addition to an initial position and discretisation, the flood fill process relies on a way to find out whether or not a voxel belongs to the domain that is going to be filled. The hit test algorithm used for this purpose is described in section 2.5.3.

For simplicity, the toolkit does not force overlap detection so it may happen that **IceParticles** overlap as the result of the geometry assembly process described in section 2.3, which results in overlapping sub-volumes. Since the flood fill algorithm keeps track of all visited voxels, no same voxel will be filled twice, but the flood fill algorithm stops walking as soon as it hits a visited voxel. When two sub-volume overlap, this leads to a premature stop. As a result, there might be unfilled voxels left. The solution to this problem is to use a secondary fill matrix for the current sub-volume, and use that matrix to determine whether or not to move to the next voxel. By resetting the content of this matrix before visiting next sub-volume, the algorithm will visit all voxels within that sub-volume.

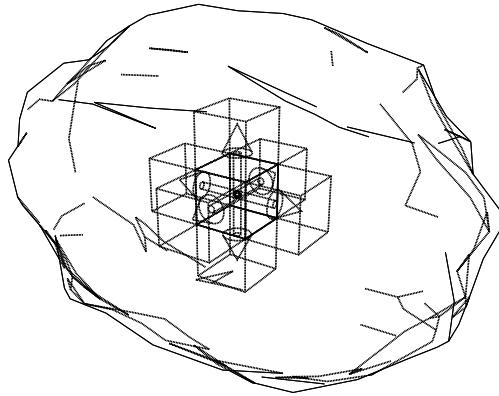


Figure 2.4: An illustration of 6-directional flood fill. Up to six neighbouring voxels are visited and filled for each voxel, as long as the next voxel is unvisited and its position is within the boundaries of the surrounding volume.

2.5 Utility routines

The toolkit provides some utility routines that are needed to make it easier to implement snowflake generating models. Since the toolkit does not use inversion of control, it is up to the user to call them when the simulation requires the corresponding functionality.

2.5.1 Face normal calculation

The face normals are calculated by taking the vector cross product between two of the vectors forming the corresponding face triangle:

$$\mathbf{n} = \overrightarrow{V_0V_1} \times \overrightarrow{V_0V_2}$$

where it is assumed V_0 , V_1 , and V_2 are oriented counter-clockwise. Some transformations may have a negative determinant and as a consequence, they flip the face normals. To fix this issue, there is a function that flips all face normals by swapping two vertex references for each face in the affected **Solid**. This works since $-\overrightarrow{V_0V_1} = \overrightarrow{V_1V_0}$.

2.5.2 Aligning normal vectors

Instead of giving rotation angles manually, it is possible to construct a rotation matrix by aligning the normal vectors of two faces so they become parallel to each other. The formula used for rotating the vector $\hat{\mathbf{u}}$ onto $\hat{\mathbf{v}}$ is

$$\mathbf{R} = \mathbf{I} + \mathbf{S} + \mathbf{S}^2 \frac{1 - \hat{\mathbf{u}} \cdot \hat{\mathbf{v}}}{|\mathbf{s}|^2}$$

where

$$\mathbf{s} = \hat{\mathbf{u}} \times \hat{\mathbf{v}}$$

and

$$\mathbf{S} = \begin{bmatrix} 0 & -s_3 & s_2 \\ s_3 & 0 & -s_1 \\ -s_2 & s_1 & 0 \end{bmatrix}$$

In the trivial case when $\mathbf{u} = \mathbf{v}$, \mathbf{R} is set to the identity matrix \mathbf{I} . If $\mathbf{u} = -\mathbf{v}$, there is a problem since the cross product becomes zero, and using the identity matrix would rotate in the wrong direction. Instead, \mathbf{R} is set to the mirroring matrix $-\mathbf{I}$.

2.5.3 Testing whether or not a point is inside a domain

The test used to determine whether or not a voxel is inside a sub-volume is illustrated in fig. 2.5. The test uses the sign of the dot product of the vector from the current point to the midpoint of the current face, and the current face normal. If the dot product is negative as happens in fig. 2.5b, then the point cannot be inside the sub-volume, and no more tests have to be performed. If it is positive for all faces as in fig. 2.5a, the point is inside the sub-volume. The test relies on the fact that all sub-volumes are convex sets, which is ensured by the definition of a sub-volume found in section 2.2.

2.5.4 Centroid and volume calculation

The volume of a **Solid** is defined as the sum of the volumes of all consisting sub-volumes. This makes the volume calculation correct if and only if there is no overlap between any of the sub-volumes. The centroid is computed as the mean positions of the centroids for the sub-volumes. Both the centroid calculation and volume calculation uses the formula found in Nürnberg, R. (2013). These formulæ are derived from the divergence theorem. For the expression for the centroid, see (Nürnberg, R. 2013). For the volume V_Σ of a sub-volume Σ it holds that¹

$$V_\Sigma = \int_{\Sigma} 1 \, dV = \int_{\Sigma} \nabla \cdot \left(\frac{1}{3} \mathbf{r} \right) \, dV = \frac{1}{3} \int_{\partial\Sigma} \mathbf{r} \cdot \, d\mathbf{S}$$

Since the surface consists of N triangles

$$V_\Sigma = \sum_{k=0}^{N-1} \frac{1}{3} \int_{T_k} \mathbf{r} \cdot \, d\mathbf{S}_k$$

¹This calculation does not require convexity, but works for any polyhedron

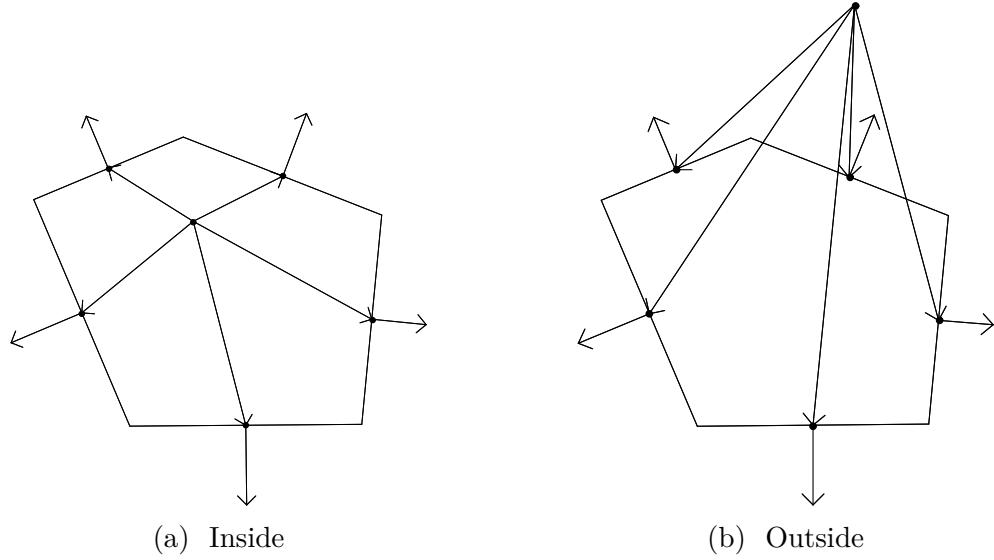


Figure 2.5: An illustration of the criterion used to test whether or not a voxel is inside a sub-volume. When a point is inside the sub-volume as in case (a), the dot product of any face normal and the vector from the point to the midpoint of the face, always has the same sign. This does not hold when the point is outside as in case (b).

For a triangle, it holds that the vector area $d\mathbf{S}_k = \frac{1}{2}\mathbf{n}_k$, where \mathbf{n}_k is the non-normalised normal vector to the triangle k . The projection of \mathbf{r} on \mathbf{n}_k is constant (Nürnberg, R. 2013), and

$$V_{\Sigma} = \frac{1}{6} \sum_{k=0}^{N-1} \mathbf{v}_o^k \cdot \mathbf{n}_k$$

where \mathbf{v}_o^k is one of the vertices in triangle k .

2.5.5 Overlap detection

Two **Solids** A and B overlap each other if and only if at least one of the following statements is true

- 1 At least one vertex of A falls inside B
- 2 At least one vertex of B falls inside A
- 3 At least one of the faces of A crosses at least one of the faces of B

Conditions 1 and 2 are checked by using hit-testing algorithm described in section 2.5.3 on the sub-volumes in A and B .

Condition 3 is checked by a triangle intersection test based on Möller, T.'s (1997). Consider the triangles T_1 and T_2 , being subsets of the planes π_1 and π_2 respectively.

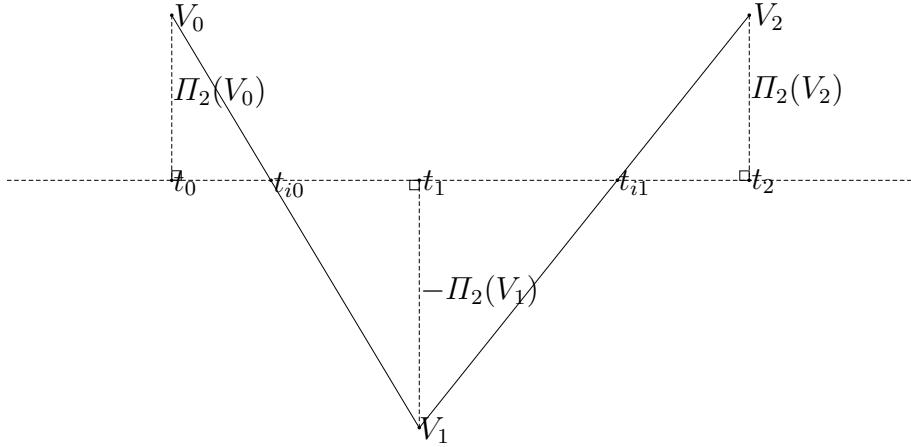


Figure 2.6: The construction used to find the parameter values t_{i0} and t_{i1} for eq. (2.3). The vertices of the triangle T_1 projected onto the plane orthogonal to π_2 . The triangle edges and the line given by eq. (2.3) forms two pairs of similar triangles.

If all vertices of T_1 lay on the same side of π_2 , or all vertices of T_2 lay on the same side of π_1 , the triangles do not intersect. If the vertices of T_1 all belong to π_2 —or equivalently—the vertices of T_2 all belong to π_1 , T_1 and T_2 are coplanar, which is also treated as non-intersecting case, since the triangles did not cross each other. Otherwise, intersection is detected by the existence of a segment of the intersection line between π_1 and π_2 , contained in both T_1 and T_2 .

The equation for the planes π_1 and π_2 is found directly from an arbitrary vertex of the triangle—take V_0^1 for π_1 —and the face normal. The equation for π_1 becomes

$$\Pi_1(P) = \hat{\mathbf{n}}_1 \cdot \overrightarrow{P_0 P} - \hat{\mathbf{n}}_1 \cdot \overrightarrow{P_0 V_0^1} = 0$$

where P_0 is an arbitrary reference point. Similarly, the equation for π_2 becomes

$$\Pi_2(P) = \hat{\mathbf{n}}_2 \cdot \overrightarrow{P_0 P} - \hat{\mathbf{n}}_2 \cdot \overrightarrow{P_0 V_0^2} = 0$$

Extending the definition of the function Π to cover the whole space, the sign of $\Pi(P)$ determines which plane side the point P is located at. The magnitude determines the orthogonal distance between P and the plane π . The point on π , where the line orthogonal to π through P intersects π , is called P_\perp .

The line of intersection between the two planes is orthogonal to both \mathbf{n}_1 and \mathbf{n}_2 . Therefore, the direction \mathbf{d} of the intersection line is determined by the vector cross product between \mathbf{n}_1 and \mathbf{n}_2 . If O is located at the line, any point P_l on it satisfies

$$P_l = t\mathbf{d} + O \quad (2.3)$$

The intersection point I between the intersection line, and one of the T_1 edges that crosses π_2 , gives the range for the parameter t for which P_l is inside the triangle.

Assume that the edge connects V_0 and V_1 . In Möller, T. (1997), the intersection point is found by projecting V_0 and V_1 on to the intersection line and observing that $\triangle V_0 P_{0\perp} I$ and $\triangle V_1 P_{1\perp} I$ are similar, and so are their projection onto the plane orthogonal to π_2 . From fig. 2.6,

$$\frac{t_{i0} - t_0}{\Pi_2(V_0)} = -\frac{t_1 - t_{i0}}{\Pi_2(V_1)} \iff t_{i0} = \frac{t_0 \Pi_2(V_1) - t_1 \Pi_2(V_0)}{\Pi_2(V_1) - \Pi_2(V_0)} \quad (2.4)$$

and

$$\frac{t_2 - t_{i1}}{\Pi_2(V_2)} = -\frac{t_{i1} - t_1}{\Pi_2(V_1)} \iff t_{i1} = \frac{t_1 \Pi_2(V_2) - t_2 \Pi_2(V_1)}{\Pi_2(V_2) - \Pi_2(V_1)} \quad (2.5)$$

where

$$t_k = \mathbf{d} \cdot (V_k - O) \quad k \in \{0, 1, 2\}$$

Applying eqs. (2.4) and (2.5) on the triangles T_1 and T_2 , gives two line segments $L_1 = \overline{t_{i0}^1 t_{i1}^1}$ and $L_2 = \overline{t_{i0}^2 t_{i1}^2}$. An overlap is then absent if and only if $L_1 \cap L_2 = \emptyset$ for all possible pair of triangles.

3 Models describing snowflake formation

This chapter describes different models that can be used to assemble an ice particle. All models that involve stochastic processes uses the `std::mt19937` random generator found in the standard library in C++11 (see (cppreference.com 2015)).

3.1 A static “model”

To be able to generate DDA data for known crystal shapes, a static model has been implemented. In this “model”, an aggregate is loaded as a graph according to fig. 2.2 from a user specified file, and a polygon mesh is assembled by traversing the graph. The aggregate file specifies parameters that can be set to affect the shape of the resulting aggregate. The aggregate file also refers to ice crystal prototype files which contain the polygon mesh for the individual ice particles that the aggregate is made of. A description of these kinds of files can be found in appendix D.

3.2 A stochastic collision model (Model A)

The model described in this section is discussed by Maruyama, K. et al. (2005), and is based on the Gillespie method¹ developed for coalescing water droplets. The method, illustrated in fig. 3.1 can be outlined as follows

- 1 Initialise the system with particles of different sizes
- 2 Choose time-step and a pair of particles (k, l) , $k < l$, for the next coalescence
- 3 Update the current time with the time-step chosen time-step
- 4 Merge the selected pair
- 5 Remove the two original particles
- 6 If the stop condition is not fulfilled go to step 2

¹Gillespie has formulated a few slightly similar algorithms. The particular variant referred to by Maruyama, K. et al. (2005) is described in (Gillespie, D. T. 1975)

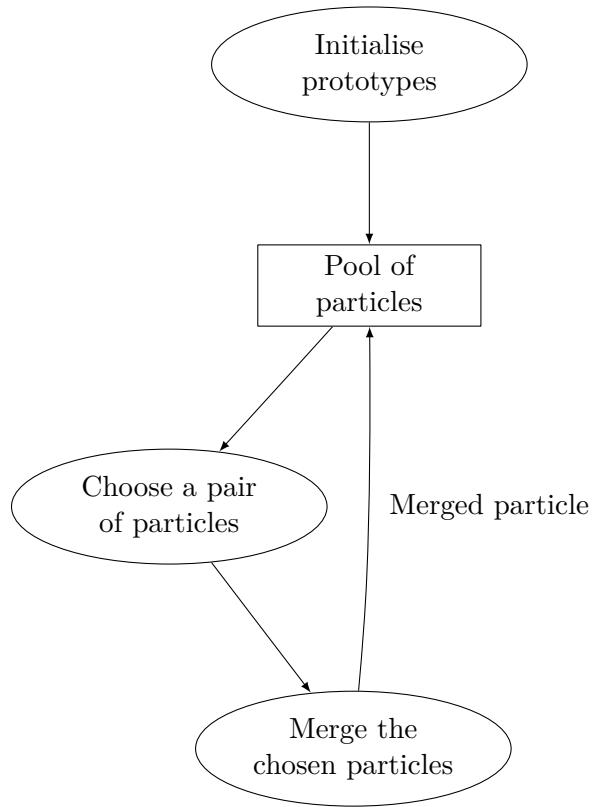


Figure 3.1: The data flow in model A. For each turn, a larger particle is created, that replaces the two smaller particles it was created from.

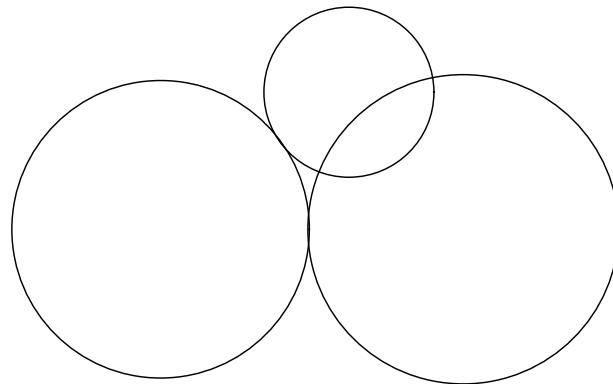


Figure 3.2: The merging scheme used in model A, without overlap detection. It is possible that merging two ice particles results in an overlap, despite the fact that the two colliding particles are attached face to face.

The model does only consider spherical crystal prototypes, which are merged so that they touch each other in a random direction like in fig. 3.2. It is not clear whether or not the possibility of overlap is taken into account.

The initial particle sizes are drawn from a gamma distribution. The time-steps are drawn from an exponential distribution with parameter C_0 , where C_0 is the probability that any particle collide within the time-step. The probability to choose a specific pair (k, l) is

$$C_{kl} = \varepsilon \frac{\sigma_{kl} |\mathbf{v}_k - \mathbf{v}_l|}{V} \quad (3.1)$$

where ε is the coalescence efficiency, σ_{kl} is the cross-section, \mathbf{v} is the particle velocity, and V is the volume of space. This equation basically says that two particles with the same velocity never collides, and the larger volume, the lower collision frequency.

The efficiency is assumed to be one, and the cross-section is assumed to be a circle with its radius equal to the sum of the maximum distance between the centre of gravity and the edge, computed for both particles. The velocity is modelled by using the particle terminal velocity, which is given by flow and fluid parameters like Reynolds number and air viscosity η . More precisely, the formula for the terminal velocity is

$$v = \frac{\eta \text{Re}}{2\rho_a} \sqrt{\frac{\pi}{A}} \quad (3.2)$$

where ρ_a is the ambient density, and A is the area of the minimal circle covering the entire ice particle, and whose normal vector is parallel to the falling direction. The expression for the Reynolds number Re is

$$\text{Re} = 8.5 \left(\sqrt{1 + 0.1519 \sqrt{\frac{8mg\rho_a}{\pi\eta^2} \left(\frac{A}{A_e} \right)^{1/4}}} - 1 \right)^2$$

where A_e is the area of the ice particle projected to the plane with normal in the falling direction, which is assumed to be random.

3.2.1 Implementation in this toolkit

Rather than computing A_e the simpler expression for the ratio between A and A_e is used, namely

$$\frac{A}{A_e} = \frac{4m}{\pi \rho R_{\max}^3}$$

where R_{\max} is the largest distance from the particle centroid to its edge. This expression is also found in Maruyama, K. et al. (2005).

The particles to merge are chosen using “full conditioning” as described by Gillespie, D. T. (1975). The strategy for this method is first to choose one particle k from

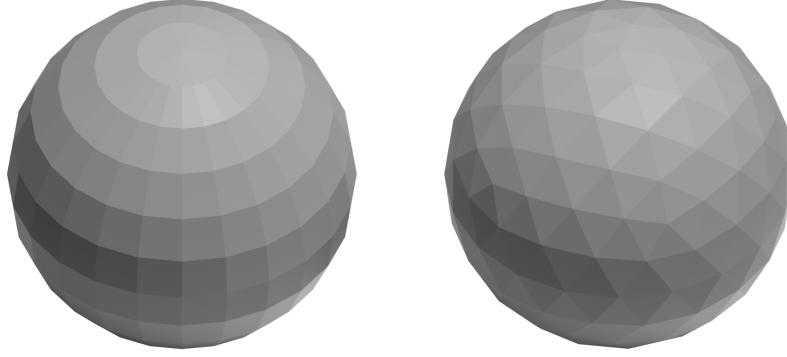


Figure 3.3: A comparison between an UV-sphere (left) and an icosphere (right). The icosphere has equal area for all of its faces but the UV-sphere has very small faces near the poles.

the probability distribution

$$P_k = \frac{1}{A} \sum_{l=k+1}^{N-1} C_{kl}$$

and then to choose particle l from the distribution

$$P_l = \frac{1}{B} C_{kl}, \quad k + 1 \leq l < N$$

where A and B are normalisation constants. This strategy results in a time processing time complexity that is $\mathcal{O}(N)$. The matrix C_{kl} , which has a number of elements that is $\mathcal{O}(N^2)$, either needs to be computed for each iteration, or stored in memory. Otherwise, the normalisation constants cannot be known. What is most beneficial depends on N , and how complicated it is to compute C_{kl} . If N is too large, the data would not fit in memory, and if C_{kl} is complicated, it takes time to compute all probabilities when computing the sum. Since N will always decrease in this model, it is waste of space to allocate a large matrix, and therefore the matrix elements are recomputed in each iteration.

Since the toolkit uses polygon meshes for representing the geometry, the randomized direction is most easily implemented by drawing one face from each of the chosen particles uniformly, and aligning their normal vectors. To locate the faces, their midpoints are attached to each other. This strategy is only correct if all available faces have the same area. However, since the sphere in the crystal library is an icosphere (see fig. 3.3), this requirement is fulfilled. Also the directions become quantised since the offset vectors can only point to the midpoint of a face. This issue is solved by the fact that the sphere consists of a large number of faces.

3.3 Non-spherical crystals and overlap detection

One can argue that ice particles seldom are spherical and that ice particles should not overlap. Also a typical snowflake tends to be more flat than the expected output of isotropic sticking probability. This motivates the formulation of a modified model, that takes these effects into account.

3.3.1 Non-spherical particles

When going from spherical to non-spherical particles, the faces no longer need to have the same area. Also, the area of each face may be non-negligible. Thus, the two issues in the implementation described in section 3.2.1 need to be fixed. The use of non-spherical particles also introduces an additional degree of freedom, since they do no longer need to have rotational symmetry around their upward direction.

Compensating for different face area

To compensate for the difference in area among the faces, the area of each face is used to determine its probability of being chosen. Since the faces are triangles, their respective area is half the magnitude of their non-normalised normal vectors (see section 2.5.1 for definition and calculation of these vectors). The probability to choose a particular face m is then

$$P_m^{(i)} = \frac{1}{2A_{\text{tot}}} |\mathbf{n}|$$

where A_{tot} is the total surface area of the corresponding chosen particle.

It is also possible to add anisotropy by weighting the face area with the magnitude of the projection of the face normal onto a preferred direction $\hat{\mathbf{w}}$. By introducing a new normalisation constant A' , the probability distribution then becomes

$$P_m^{(a)} = \frac{1}{A'} |\hat{\mathbf{w}} \cdot \hat{\mathbf{n}}| |\mathbf{n}| = \frac{1}{A'} |\hat{\mathbf{w}} \cdot \mathbf{n}| \quad (3.3)$$

If there still should be a probability to choose a direction orthogonal to $\hat{\mathbf{w}}$, a blending parameter can be used to blend between a complete isotropic probability by introducing a blending parameter $\alpha \in [0, 1]$. This gives the probability

$$P_m = \alpha P_m^{(a)} + (1 - \alpha) P_m^{(i)}$$

Since the toolkit stores all ice particles as a collection of sub-volumes, the face selection has to be implemented by first picking a sub-volume, and then by picking a face from that sub-volume. The choice of sub-volume is done by using probabilities

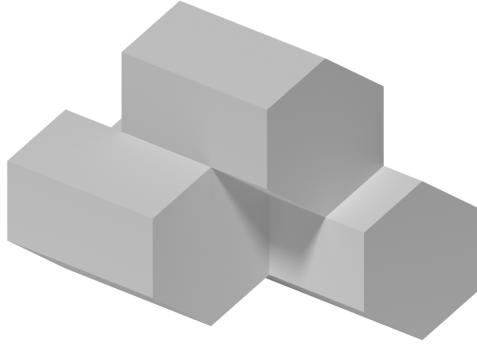


Figure 3.4: An example of a “timber stack”. This may happen if ice particles are merged using the same yaw angle.

computed from the total area of all their visible faces. When computing the total area, it is possible to use eq. (3.3) to add anisotropy like

$$A_{\text{tot}} = \frac{1}{2} \sum_{k=0}^{N-1} |\hat{\mathbf{w}} \cdot \mathbf{n}_k|$$

where the sum is computed over all visible faces in a specific sub-volume.

Dealing with non-negligible face area

Since the face area may no longer be negligible, a new particle can stick anywhere on the selected face. To choose another anchor point than the face midpoint, a point (ξ_0, η_0) from the triangle

$$(\xi, \eta) : \xi \in [0, 1] \wedge \eta < 1 - \xi$$

is drawn uniformly. The triangle vertices are mapped to the vertices (V_0, V_1, V_2) of the selected face. From the equation of the plane, the offset vector (see fig. 2.3) becomes

$$\mathbf{u} = V_0 + \xi_0(V_1 - V_0) + \eta_0(V_2 - v_0) - P_0 \quad (3.4)$$

The equation for \mathbf{v} is similar.

Dealing with non-symmetric ice particles

The final degree of freedom is resolved by choosing the yaw angle in the rotated coordinates from a uniform distribution $U(0, 2\pi)$. This is needed, otherwise the simulation may result in a lot of “timber stacks” (see fig. 3.4). The angle can also be quantised to integer multiples of $\frac{\pi}{3}$, since this angle matches the typical crystal structure of ice.

3.3.2 Overlap detection

To avoid the possibility of particle overlap illustrated in fig. 3.2, an algorithm for detecting whether or two particles overlap is needed. Also, there has to be strategy for dealing with events that would lead to an overlap. The overlap is detected by using the algorithms described in section 2.5.5. When an overlap occurs, the event is simply dropped, meaning that steps 4 and 5 in the algorithm outlined in section 3.2 are skipped.

3.4 Adding more processes to the model (Model B)

In a real cloud, there are at least three more processes in addition to coalescence: melting, particle drop-out, and new particle formation. Introducing the possibility of adding new particles to the cloud adds the possibility to keep a certain number of particles inside the cloud so the process can go on forever. The two other processes have an impact on the size distribution of the particles in the cloud: No drop-out or melting and the particle sizes would keep growing as long as new particles are added. This is because without drop-out or melting, the only process that reduces the number of particles is the coalescence process, which is responsible for creating larger particles. With all these processes, the data flow for this model will look like the one in fig. 3.5.

When constructing this extended model—this must also be true for model A, it is assumed that every spatial location within the cloud could be considered identical with respect to quantities affecting the transition rates between different states in water. An example of a scenario satisfying this criterion is when pressure and temperature is roughly constant.

3.4.1 Event probabilities

Equation (3.1) for the coalescence probability is inherited from model A, as well as the random falling direction, which is equivalent to random orientation. However, since three more processes are added, there are three new parameters. To keep the number of parameters graspable, the model for the terminal velocity (eq. (3.2)) is replaced by the value given by the basic drag formula

$$F_{\text{drag}} \propto \rho_a A_k |\mathbf{v}_k|^2$$

where ρ_a is the ambient density, A_k is an area for particle k , and \mathbf{v}_k is the velocity of particle k . Now

$$A \rho_a |\mathbf{v}_k|^2 \propto V_k \rho g \iff v \propto \sqrt{\frac{V_k \rho g}{\rho_a A_k}} = C \sqrt{\frac{V_k}{A_k}} \quad (3.5)$$

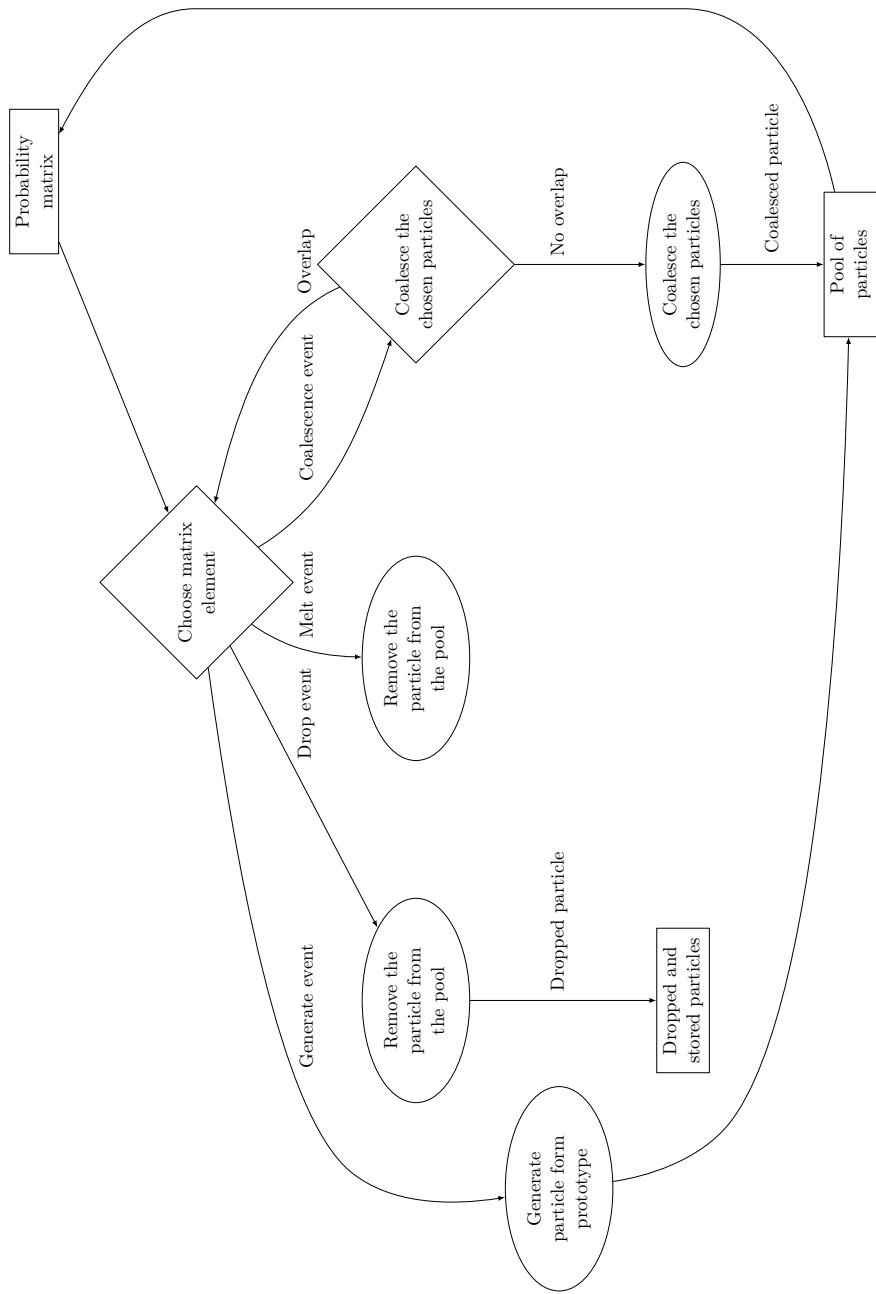


Figure 3.5: The data flow in model B. The addition of additional processes introduces two branches. The addition of dropping particles (as opposed to melting them) requires a new pool of particles. The reason for storing these non-active particles rather than discard them, is that they can be used for further analysis.

where ρ is the particle density and g is the gravity of earth. The area A_k of particle k is computed as the square of the maximal distance from the centroid of particle k . This simpler model has one parameter while the model used by Maruyama, K. et al. (2005) has five. When the formula is inserted into other expressions, it will be clear that the parameter C can be eliminated as well, leaving no free parameters.

It may seem like the simple formula in eq. (3.5) leaves too many details, since there is only one parameter, but there is no guarantee that a more complicated expression cover all cases equally well. For the terminal velocity, the particle geometry will affect the flow, and the geometry of the particles changes after coalescence events, and then it is likely that the model that worked well for the previous case is no longer valid. In particular, the drag coefficient, which have been set to a constant eq. (3.5), depends on the characteristics of the flow, which in turn is dependent on the particle geometry.

Melting and particle formation

A simple model for melting is to assume that a particle receives constant power from its neighbourhood. Then, the time it takes before the particle k has melted is proportional to its latent heat, which for a homogeneous material is proportional to the particle volume. The probability for the particle to melt during a time-step then becomes

$$P_{mk} = \frac{R_m}{V_k} \quad (3.6)$$

where V_k is the particle volume and R_m is proportional to the specific latent heat of the particle. Other quantities that are likely to affect R_m , include the cloud temperature and the total surface area of the particle. Even though the latter definitely is dependent on the choice of particle, and as a consequence cannot be hidden within R_m , this model does not include that effect. When building a more elaborate model, this effect should probably be included.

New particles are added to the cloud at a constant rate R_g from an inexhaustible source of ice particles. This makes it possible to start with zero particles, and it also makes it possible to maintain a cloud for an arbitrary number of iterations.

When a particle melts in the model, it water content disappears with it, so it would be more correctly named sublimation. If the particle water content stays, there is an equilibrium process



The reaction rates are affected by atmospheric pressure and temperature. The

corresponding differential equations (with source terms added) become

$$\begin{cases} \frac{d}{dt}[\text{H}_2\text{O}(\ell)] = -C_1[\text{H}_2\text{O}(\ell)] + C_2[\text{H}_2\text{O}(s)] \\ \frac{d}{dt}[\text{H}_2\text{O}(s)] = C_1[\text{H}_2\text{O}(\ell)] - C_2[\text{H}_2\text{O}(s)] + S \end{cases}$$

where $[\text{H}_2\text{O}(\ell)]$ is the concentration of liquid water, $[\text{H}_2\text{O}(s)]$ is the concentration of solid water, and S is a source term. The eigenvalues λ for the homogeneous system are given by the equation

$$(-C_1 - \lambda)(-C_2 - \lambda) - C_1 C_2 = 0 \iff (C_2 + C_1)\lambda + \lambda^2 = 0$$

The eigenvalues are $\lambda_1 = 0$ and $\lambda_2 = -(C_1 + C_2)$, and the system has a stable manifold that is the line

$$\frac{[\text{H}_2\text{O}(s)]}{[\text{H}_2\text{O}(\ell)]} = \frac{C_1}{C_2}$$

Since the amount of liquid water is not used in any other process than ice particle formation and melting, the equilibrium process described by eq. (3.7) only adds a time delay, and reduces the stationary production rate of ice. Thus the model without the equilibrium is equivalent with an equilibrium process with a large value of $|\lambda_2|$.

Particle dropout

The probability for a particle to leave the cloud—particle drop-out—is assumed to be proportional to the particle speed $|\mathbf{v}_k|$, giving

$$P_{dk} \propto |\mathbf{v}_k|$$

The only particles that are considered for leaving during one time-step are those that are located at the surface of the cloud. Let N_{surf} be the number of particles at the surface of the cloud, and r be some equivalent radius of the cloud. For a homogeneous cloud, the number of particles N in the cloud has to be proportional to the volume of the cloud, which is proportional to r^3 and

$$N \propto r^3 \tag{3.8}$$

Also, N_{surf} is proportional to surface area of the cloud, which is proportional to r^2 . Therefore

$$N_{\text{surf}} \propto r^2$$

Substituting r given by eq. (3.8),

$$N_{\text{surf}} \propto N^{2/3}$$

$$N_{\max} + 1 \text{ rows} \left(\begin{array}{ccccc} 0 & 3 & \cdots & 3 & 2 \\ 3 & 0 & \ddots & \vdots & 2 \\ \vdots & \ddots & \ddots & 3 & \vdots \\ 3 & \cdots & 3 & 0 & 2 \\ 1 & 1 & \cdots & 1 & \times \end{array} \right) \underbrace{N_{\max} + 1 \text{ columns}}_{\boxed{\begin{array}{ll} 0 & \text{New particle formation} \\ 1 & \text{Particle drop-out} \\ 2 & \text{Melting} \\ 3 & \text{Coalescence} \\ \times & \text{Not assigned} \end{array}}}$$

Figure 3.6: The layout of the probability matrix used by model B. Different digits marks the locations for the probabilities of different kinds of events. When an element which indicates that an event of type 3 is to occur, the row and column number defines which two particles that will coalesce. To determine the affected particle for particle drop-out (type 1) and melting (type 2), the column and row numbers respectively are used. The probability for new particle formation (type 0), the sum of the diagonal elements.

For the probability that any particle P_d will drop-out is then proportional to N_{surf} . Provided that the speed has an expectation value,

$$P_d \propto \sum_{\text{Surf}} |\mathbf{v}_k| \sim N^{2/3} \mathbb{E}|\mathbf{v}_k| \quad N \rightarrow \infty$$

Dividing the asymptotic limit by the total number of particles now gives an expression for P_{dk} :

$$P_{dk} = \frac{R_d}{N^{1/3}} |\mathbf{v}_k| \quad (3.9)$$

where R_d is the event rate.

As mentioned above, eq. (3.1) for the coalescence probability, is the same as before. By observing that all probabilities that involve the terminal velocity always scales linearly with the speed, the proportionality constant C in eq. (3.5) can be set to 1. The same goes for V in eq. (3.1), which only affects the time-scale of the simulation.

3.4.2 Storing probabilities

The number of probabilities, required by both this model and the two previous models, scales by N^2 . When the number of particles always decreases between two iterations as it does in the previous model, it makes sense not to store all probabilities since it makes it possible to start with a larger number of particles, but when the number of particles remains large during the simulation, *profiling* showed that computing C_{kl} is expensive, and therefore, the method of storing C_{kl}

was chosen. This way, only those elements that are affected by an event need to be recomputed, which—ignoring the computation of the matrix sum for the time being—reduces the computational complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$. On the other hand, it increases the memory usage from $\mathcal{O}(N)$ to $\mathcal{O}(N^2)$, and—if not considering memory reallocation—it also sets an upper limit on how many particles the current simulation can deal with. It should be noted that it in theory is sufficient to store P_k , by subtracting and adding values to the sum for each iteration. However, this approach may accumulate errors and has therefore not been chosen.

With precomputed matrix elements, most of the simulation time is spent in computing the matrix sum, which still is $\mathcal{O}(N_{\max}^2)$, but each operation in computing the matrix sum is much cheaper than computing a matrix element. To increase performance, this part of the simulation is both multi-threaded, and vectorised. This implementation puts a restriction on N_{\max} : $N_{\max} + 1$ has to be divisible by both four—due to the use of *Advanced Vector Extensions* (AVX)-256 with **doubles** (Intel Corporation 2015), and the number of computation threads. It would have been possible to remove these constraints, but it would have required more code, reducing the performance for the optimal case.

The probabilities are stored in an $(N_{\max} + 1) \times (N_{\max} + 1)$ matrix \mathbf{E} illustrated in fig. 3.6. All off-diagonal elements except the last row and column store the coalescence probability between different particles. The last row stores the probabilities of particle drop-out and the last column stores the probabilities of melting. The main diagonal is used to store the probability of new particle formation. The reason for splitting the probability for new particle formation is to reduce the effect of possible truncation when normalising the matrix.

Since the probability for new particle formation has been distributed along N entries in the main diagonal of \mathbf{E} , the corresponding matrix elements have to be divided by N . The melting and drop-out probabilities are unmodified and given by eq. (3.6) and eq. (3.9) respectively. The number of elements that results in a coalescence event for a given particle is twice as large as the actual number of possible coalescence events. This will bias the event selection towards coalesce. To make the selection unbiased, all other elements has to be multiplied by two.

Throughout the simulation, the size of \mathbf{E} is never changed. When a particle leaves the system, all of the corresponding matrix elements in \mathbf{E} are set to zero, effectively removing the possibility to choose that particle. That is, if particle k is removed, all values in row and column k are set to zero except element (k, k) , which being a growth probability is not associated with any particle.

3.4.3 The resulting algorithm

The algorithm for implementing the extended model looks similar to the algorithm for model A, but with more choices inside the loop, since the new model has melting, generation and drop-out processes in addition to the coalescence process. Also, if

the event type is a coalescence event that would result in overlap, the event is rejected and a completely new event is to be drawn. There are also some additional initialisation that has to be done, given that there is now a possibility to choose different crystal prototype, and the fact that probabilities are stored in a matrix. With all steps, the algorithm becomes as follows:

- 1 Load the desired crystal prototype
- 2 Allocate the matrix \mathbf{E} of size $(N + 1) \times (N + 1)$
- 3 Allocate an array A consisting of N **IceParticles**.
- 4 Choose an matrix element (k,l) from \mathbf{E}
- 5 Now, if
 - 1 $k = l$ (This is a new particle generation event):
 - 51 Find a “dead” g **IceParticle** in A .
 - 52 If there is such g
 - 521 Create a new **IceParticle** in the place of g , with the desired size
 - 522 Update the affected matrix elements
 - 2 $k = N$ (This is a particle drop-out event):
 - 51 Save the affected grain to a dynamic array for later use
 - 52 “Kill” the grain **IceParticle** located at index l in A
 - 53 Update the affected matrix elements
 - 3 $l = N$ (This is a particle melt event):
 - 51 “Kill” the grain **IceParticle** located at index k in A
 - 52 Update the affected matrix elements
 - 4 $k \neq l \wedge k < N \wedge l < N$ (This is a coalescence event):
 - 51 Choose a face f_k from the **IceParticle** g_k located at position k in A
 - 52 Choose a face f_l from the **IceParticle** g_l located at position l in A
 - 53 Choose offset vectors \mathbf{u} and \mathbf{v} (see eq. (3.4) and fig. 2.3), so their end lies on f_k and f_l respectively
 - 54 Calculate a rotation matrix \mathbf{R} such that the normal vectors of f_k and f_l would become anti-parallel if one of the two **IceParticles** were rotated by \mathbf{R} . Assume g_l is rotated.
 - 55 Rotate g_l around its new heading

- 56 If merging g_l into g_k does not result in overlap
 - 561 Merge g_l into g_k
 - 562 “Kill” g_l
 - 563 Update affected matrix elements
- 6 Update current simulated time, given the probability that any event happened
- 7 If stop condition is not fulfilled, go to step 4

4 Evaluating models

To evaluate models, a set of measurable quantities has to be computed. To understand the system behaviour it is interesting to measure how quantities vary with time. However, many studies of real snowflakes focus on geometrical aspects (Baran, A. J. 2012, Garrett, T. J. et al. 2015, Matrosov, S. Y. 2007, Mitchell, D. L. et al. 1990), rather than time evolution. Therefore, the analysis has focused on such aspects. The aspects analysed are **the relation between particle volume and radius**, **particle size distribution**, and **the spherical volume fill ratio**.

4.1 The relation between particle volume and radius

The relation between particle mass m and its maximal radius R_{\max} measured from the centre of the particle is expected to look like

$$m = CR_{\max}^{\beta} \quad (4.1)$$

where β , also known as the *fractal dimension*, is from theory expected to be equal to two (Stein, T. H. M. et al. 2015). This value is also supported by experiments (Matrosov, S. Y. 2007). Since constant ice density is assumed, eq. (4.1) is equivalent to

$$V = \kappa R_{\max}^{\beta} \quad (4.2)$$

where $\kappa = \frac{C}{\rho}$. Equation (4.2), with particular values of κ and β is expected to hold for a specific set of model parameters. This means that given a particular set of model parameters, the value of κ and β should be the same. However, there is no assumption that there is any relation between κ and β .

The fractal dimension β is a measurement of the effective number of directions in which the particles grow. If $\beta = 1$, the particle is formed along a curve, and $\beta = 2$ implies that the particle is formed on a surface. The constant κ is a *volume growth coefficient* and gives a measure of how much the volume of a particle increases when looking at larger particles.

A value for C has been measured for a set of natural particles by Mitchell, D. L. et al. (1990). The value given in that article is 0.088 kg/m^2 . Other possible values are 0.12 kg/m^2 and 0.11 kg/m^2 (Matrosov, S. Y. 2007). To be able to compare the data retrieved from simulations with this value, eq. (4.1) needs to be normalised. Assuming regular solid ice, $\rho = 917 \text{ kg/m}^3$. To get rid of the length unit, an estimate

of the expectation value r_{\max} of the crystal prototypes used to generate the particles is used. By multiplying κ by $r_{\max}^{\beta-3}$, the dimensionless *normalised volume growth coefficient* α is constructed and

$$\alpha = \kappa r_{\max}^{\beta-3} = \frac{Cr_{\max}^{\beta-3}}{\rho} \quad (4.3)$$

From Schmitt, C. G. et al. (2014), $r_{\max} \approx 0.1 \text{ mm} = 1 \times 10^{-4} \text{ m}$. This value has been retrieved by measuring the diameter of a crystal in an image—more precisely the first image in figure 2 in Schmitt, C. G. et al. (2014)—captured using a *Cloud Particle Imager* (CPI) probe. Combining the data retrieved from Mitchell, D. L. et al. (1990) and Schmitt, C. G. et al. (2014) gives $\kappa = 9.6 \times 10^{-5} \text{ m}$, and $\alpha = 0.96$. The coefficients from Matrosov, S. Y. (2007) gives $\alpha = 1.2$ and $\alpha = 1.3$ respectively for $\beta = 2$.

4.2 The particle size distribution

The particle sizes have been found by to follow an exponential distribution (Garrett, T. J. et al. 2015, Matrosov, S. Y. 2007).

$$\frac{d}{dN}(R_{\max}) = C \exp(\lambda R_{\max}) \quad (4.4)$$

This equation is somewhat problematic, since the parameter λ is not dimensionless, being a reciprocal length. Therefore, this parameter needs to be normalised. By using r_{\max} , eq. (4.4) can be written as

$$\frac{d}{dN}(R_{\max}) = C \exp(\lambda R_{\max}) = A \exp\left(\lambda r_{\max} \frac{R_{\max}}{r_{\max}}\right) \quad (4.5)$$

Therefore, measuring R_{\max} in terms of r_{\max} , gives a dimensionless parameter $\gamma = \lambda r_{\max}$, and this value is used in favour of λ .

Measurements show that λ is in the range -13 mm^{-1} to -1.2 mm^{-1} (Garrett, T. J. et al. 2015, Braham, R. R. 1990), depending on snowfall intensity (Matrosov, S. Y. 2007). Together with the data from Schmitt, C. G. et al. (2014), this gives a value of γ in the range -1.3 to -0.12 . These figures can also be converted to expectation values for the distribution given by eq. (4.5). Then $\mathbb{E}\left(\frac{R_{\max}}{r_{\max}}\right)$ is in the range 2.5 to 8.3. The size distributions in Garrett, T. J. et al. (2015) have been obtained by collecting data from falling snow by using a *Multi-Angle Snowflake Camera* (MASC), and by rejecting particles with low complexity due to difficulties in the measurement procedure (Garrett, T. J. et al. 2015). Also in this project, the low-complexity particles has been rejected for analysis.

The particle size is mainly interesting because it is a measurement of complexity. Given two particles like those in fig. 4.1 that are composed of crystal prototypes

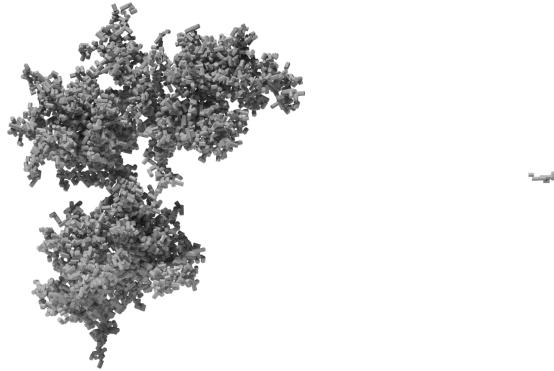


Figure 4.1: A large and a small particle. The larger particle is composed of more crystal prototypes than the smaller one.

of similar sizes, the larger particle needs to be composed of more crystal prototypes than the smaller one. While it is possible to measure complexity (Schmitt, C. G. et al. 2014), particle size is easier to define and measure, and therefore the size is used instead.

4.3 The spherical volume fill ratio

Besides the fractal dimension β and the particle size distribution, one can measure the spherical volume fill ratio v_r . This quantity is defined as ratio of the particle volume by the volume of a sphere with the same R_{\max} .

$$v_r = \frac{3\kappa R_{\max}^{\beta}}{4\pi R_{\max}^3} \quad (4.6)$$

Being a ratio between volumes, it is dimensionless and thus, this value can be compared to measurements directly, given the ratio $\frac{\kappa R_{\max}^{\beta}}{R_{\max}^3}$. It also serves as a measurement of κ in eq. (4.2) because

$$v_r = \frac{3\kappa R_{\max}^{\beta}}{4\pi R_{\max}^3} \iff \kappa = \frac{4\pi v_r R_{\max}^3}{3R_{\max}^{\beta}}$$

Since v_r is dependent on R_{\max} , it is practical to compute an averaged value v_{r0} . The most natural definition would be $v_{r0} = \langle v_r \rangle$, but this definition has problems for $\beta < 3$. Instead, v_{r0} is taken as a pseudo-expectation value, defined as

$$v_{r0} = \frac{\langle V \rangle}{\langle V_{\text{sphere}} \rangle} = \frac{3\kappa \langle R_{\max}^{\beta} \rangle}{4\pi \langle R_{\max}^3 \rangle} \quad (4.7)$$

Because κ and β is assumed to vary depending on simulation parameters, the mean values are connected a specific set of model parameters.

Both the numerator and the denominator in eq. (4.7) contain expectation values, of R_{\max} to the power of a positive constant. By using the assumption that $R_{\max} \sim \text{Exp}(-\lambda)$, it is possible to find the expectation value of R_{\max}^a . This value can then be substituted into eq. (4.7), with $a = \beta$ and $a = 3$ respectively. From the definitions of an expectation value for a continuous random variable, and the exponential distribution,

$$\langle R_{\max}^a \rangle = \int_0^\infty R_{\max}^a \cdot (-\lambda) \exp(\lambda R_{\max}) dR_{\max}$$

This integral can be simplified by the substitution

$$x = -\lambda R_{\max} \iff R_{\max} = -\frac{x}{\lambda} \implies \frac{d}{dx} R_{\max} = -\frac{1}{\lambda} \iff dR_{\max} = -\frac{1}{\lambda} dx$$

which gives

$$\langle R_{\max}^a \rangle = \frac{1}{(-\lambda)^a} \int_0^\infty x^a \exp(-x) dx = \frac{1}{(-\lambda)^a} \Gamma(a+1)$$

and from eq. (4.7)

$$v_{r0} = \frac{3\kappa\Gamma(\beta+1)}{(-\lambda)^\beta} \Big/ \frac{4\pi\Gamma(4)}{(-\lambda)^3} = \frac{3\kappa \cdot (-\lambda)^{3-\beta}\Gamma(\beta+1)}{4\pi\Gamma(4)} \quad (4.8)$$

It is now possible to construct a quantity that only depends on the dimensionless quantities $-\lambda R_{\max}$ and β . This can be done by considering the ratios $\frac{v_r}{v_{r0}}$, as well as its inverse. From eqs. (4.6) and (4.8)

$$\begin{aligned} \frac{v_r}{v_{r0}} &= \frac{3\kappa R_{\max}^\beta \cdot 4\pi\Gamma(4)}{4\pi R_{\max}^3 \cdot 3\kappa \cdot (-\lambda)^{3-\beta}\Gamma(\beta+1)} = \frac{\Gamma(4)}{R_{\max}^{3-\beta} \cdot (-\lambda)^{3-\beta}\Gamma(\beta+1)} \\ &= \frac{6}{(-\lambda R_{\max})^{3-\beta}\Gamma(\beta+1)} \end{aligned} \quad (4.9)$$

which hereafter will be called the *reduced spherical volume fill ratio*. In the special case when $\beta = 2$, it holds that $\Gamma(\beta+1) = 2$, and thus

$$\left. \frac{v_r}{v_{r0}} \right|_{\beta=2} = -\frac{3}{\lambda R_{\max}} \quad (4.10)$$

In eq. (4.10), the only parameter that depends on simulation parameters is λ . Therefore, by plotting the reduced spherical volume fill ratio as a function of $x = -\lambda R_{\max}$, all data points should end up on the line $\frac{3}{x}$.

A possible use of eq. (4.9) is that it can be used to compare different data-sets. By plotting the measured $\frac{v_r}{v_{r0}}$ against $-\lambda R_{\max}$ in a log-log plot, all points should end up on a line and therefore, it is possible to justify the claim that the particle size is exponentially distributed, and that $\beta = 2$.

4.4 Data collection procedure

The quantities recorded in order to compare models are listed in table 4.1, together with their symbol. Instead of recording the ensemble average of the particle speed and reciprocal volume, these properties are recorded for individual particles and the reciprocal volume is derived from the regular volume. This way, it is possible to generate histograms, showing the distribution of the per particle quantities. Since the toolkit works in units defined by the crystal prototypes so called system units, all size measurements are given in system units.

In addition to the particle related properties mentioned above, the simulated time and the current system time is recorded. The former can be used to check if some other quantity converge to a certain value, and the latter is used as a measure of simulation expensiveness.

In order to understand how the model parameters affects the result, all quantities are computed from runs with different combinations of parameters. At a regular iteration interval, all per-particle quantities are stored on a file for later analysis. Also, the particle independent data are written to a separate file.

4.4.1 Collection of data from model B

In model B, there are two sets of particles: those that have not yet left the cloud, and those that have. To capture all statistics, both sets are sampled individually. The model has been evaluated both with respect to event rates, and different particle geometry. The hypothesis is that the event rates mainly affect the particle size distribution, that is the size distribution parameter λ in eq. (4.4), and the geometry has a larger impact on β in eq. (4.2), although there can be a small opposite coupling as well.

To be able to compute the normalised size distribution parameter γ , the value r_{\max} in eq. (4.5) needs to be computed. This is done by computing the average of R_{\max} for all non-coalesced particles generated with the same geometry parameters, and by using the obtained value as r_{\max} .

For the simulations with varying event rates, the parameter values listed in table 4.2 are used, together with the column type crystal prototype. To see what happens when the model is applied to different geometry, the parameter values listed in table 4.3. Also for this simulation, the column type crystal prototype is used. In these simulations, the expectation value of L is chosen so that completely deterministic value of L would preserve the crystal prototype volume from the other simulation. By fixing the particle volume $\frac{3\sqrt{3}}{2}\left(\frac{1}{3}\right)^2 = V_0 = \frac{3\sqrt{3}}{2}La^2$, the axial ratio becomes

$$\frac{L}{2a} = \frac{1}{18a^3} \quad (4.11)$$

Since the model is stochastic, the same set of simulations is run twice, each time

Table 4.1: The quantities recorded during simulations. **Per particle** quantities vary between all particles within one iteration, and the **Per iteration** quantities vary only between iterations.

	Symbol	Description
Per particle	R_{\max}	The maximum distance from particle centroid to the particle edge
	L_x	The width of the particle bounding box
	r_{xy}	The x/y aspect ratio. $r_{xy} = \frac{L_x}{L_y}$ where L_x and L_y are sizes of the bounding box described in section 2.4.
	r_{xz}	The x/z aspect ratio. $r_{xz} = \frac{L_x}{L_z}$ where L_x and L_z are sizes of the bounding box described in section 2.4.
	V	The particle volume. Since the density is assumed to be constant, this quantity is proportional to the particle mass
	$ \mathbf{v} $	Particle speed
	N_p	The number of building blocks used for creating the current particle
Per iteration	N_{cloud}	The number of particles in the cloud
	N_{dropped}	The number of particles that have left the cloud
	$\sum C_{kl}$	The total coalescence rate
	τ	The simulated time
	t	The system time measured in POSIX time

Table 4.2: Parameter values used to evaluate model B. The logarithmic increments are used in order to increase the parameter ranges. The crystal prototype and its parameters are described in table C.1.

Parameter	Description	Values
Crystal prototype	Initial shape	<code>bullet.ice</code>
t	Length of tip (see table C.1)	0
a	Side of cross section hexagon (see table C.1)	0.33
L	Crystal length (see table C.1)	$\mathbb{E}(L) = 1$, $\text{std}(L) = 0.25$
R_g	Generation rate (See section 3.4)	15×10^6 , 20×10^6 , 33×10^6
R_m	Melt rate (See section 3.4)	500, 1000, 2000
R_d	Drop rate (See section 3.4)	10 000, 20 000, 50 000

Table 4.3: The parameter values used to evaluate model B with respect to different geometry. The crystal prototype and its parameters are described in table C.1. The quantities are the same as in table 4.2.

Parameter	Values
Crystal prototype	<code>bullet.ice</code>
t	0
a	0.17, 0.22, 0.28, 0.33, 0.39, 0.44, 0.5, 0.55, 0.61, 0.66
L	Random with $\text{std}(L) = 0.25 \mathbb{E}(L)$, and $\mathbb{E}(L) = \frac{0.33^2}{a^2}$
R_g	33×10^6
R_m	500
R_d	20 000

with seeds created by reading four bytes from the special file `/dev/urandom`, provided by the Linux® kernel (Linux Kernel Organization 2013).

4.5 Model parameter fitting

Parameters are found by using the least square approach on the simulation output data. That is, to find the parameters ξ that minimises

$$\delta(\xi) = \sum_{k=0}^{N-1} (y_k - f(\xi, x_k))^2$$

where x_k and y_k are values obtained from the simulation, and f is the model that describes how y depends on x . The error measure e used is the root mean square error, normalised to the full range of the observed y values. With this definition

$$e = \frac{\sqrt{\delta(\xi)}}{\sqrt{N} \left(\max_{k \in [0, N[} (y_k) - \min_{k \in [0, N[} (y_k) \right)} \quad (4.12)$$

One possible approach when solving the least square problem, is to transform the equation

$$y = f(\xi, x)$$

into a linear form

$$y' = L(\xi, x')$$

Then, the problem can be solved by non-iterative algorithms derived from matrix algebra. However, the transformation may have some undesirable effects (see Newman, M. C. (1993)) and therefore, a non-linear solver `fminsearch` (Eaton, J. W. et al. 2011) is used. This solver requires an initial guess for ξ (Eaton, J. W. et al. 2011), which is found by solving the corresponding linear problem. This is a form of *bootstrapping*.

4.5.1 Dealing with time-dependent quantities

The quantities in table 4.1 may vary between iterations. To give them a time-independent value, their behaviour for large simulated times are studied. If the value seems to converge towards a limit y_∞ —like in fig. 4.2—this value is used. Otherwise, the quantity cannot be assigned a time-independent value. To find such a limit, two models are used to described the time-evolution of the quantity. A linear model

$$y(\tau) = C\tau + m \quad (4.13)$$

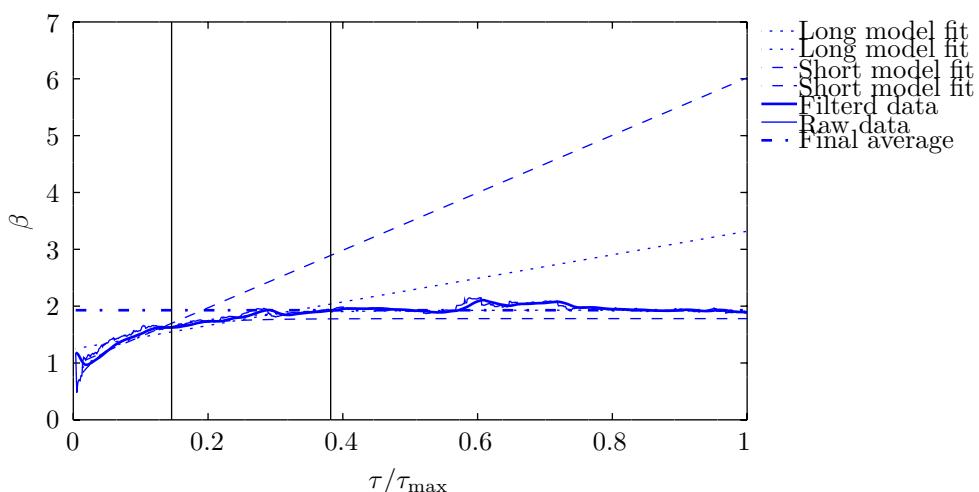


Figure 4.2: An example of the dependency between the simulated time τ and the fractal dimension β . This time-series was obtained by running model B, with $R_g = 22 \times 10^6$, $R_m = 500$, and $R_d = 20\,000$. The dashed and dotted lines are the least-square fits of the parameters in eq. (4.13) and eq. (4.14), to the data to the left of the vertical lines. When fitting to less data (dashed lines), the exponential model (eq. (4.14)) gives significantly better prediction than the linear model (eq. (4.13)). In this case, the difference is smaller for the larger data-set (dotted lines).

and an exponential model with offset

$$y(\tau) = C \exp(a\tau) + y_\infty, \quad a < 0 \quad (4.14)$$

If the linear model better determines the collected data, no good value for y has been found. If the exponential model is better, it is more likely that a limit exists. The linear model is chosen as reference because it

- does not predict a limit at infinity
- is monotonic
- has a second derivative that is not greater than zero
- is easy to work with

The comparison between the two models is based on their prediction power. To measure the prediction power, the data-set is split in two parts (the vertical line in fig. 4.2), their relation in duration being the same as the longest duration to the total duration. Those data points corresponding to the first and smaller part of the time-series are used to fit the model parameters. The remaining part is used to compute a prediction error from eq. (4.12), with the parameters computed from the data in the first part. The ratio in prediction error between the linear model for y and the exponential model for y is called the *prediction ratio* of y and is denoted $\text{Pr}(y)$. A large value of $\text{Pr}(y)$ indicates that it is more likely that a limit exists. If the prediction ratio is less than 1, the captured part of the time-series does not indicate any convergence.

Due to some apparent stability problems in fitting the exponential model eq. (4.14), the input data is first filtered. To easily preserve the initial value, and to avoid the need of interpolating data, a digitally implemented analogue filter of second order, with its initial state set to $(y(0), 0)$, is used. Calling the filter output y_f , the initial value problem for filtering the signal becomes

$$\begin{cases} \omega_c^2 y_f = \ddot{y}_f + 2\omega_c \zeta \dot{y}_f + \omega_c^2 y_f \\ y_f(0) = y(0) \\ \dot{y}_f(0) = 0 \end{cases}$$

To avoid *ringing artefacts*, a critically damped filter, that is $\zeta = 1$, is used. The cut-off frequency is set through a desired raise-time measured in units of signal duration. The initial value problem is solved with the trapezoidal rule with the time-steps given by the input data.

While fitting the parameters in the linear model can be done through the linear least squares method, the exponential model being non-linear requires bootstrapping of the non-linear solver. Since it is assumed that $y_f \rightarrow y_{f,\infty}$ when $\tau \rightarrow \infty$, the mean value of y_f computed over the largest possible τ values is used as initial guess of

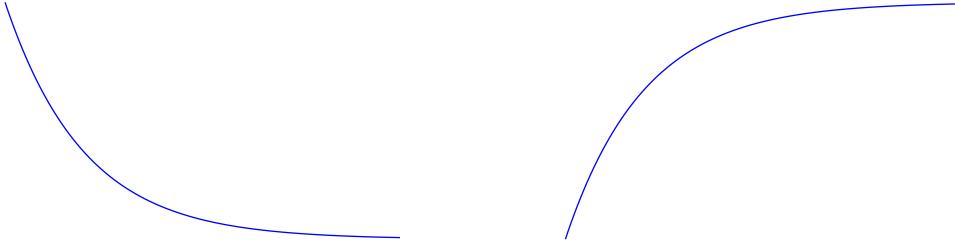


Figure 4.3: The two possible shapes of a convergent time-series.

$y_{f,\infty}$. Given an initial value for $y_{f,\infty}$, the most likely shape S (see figure fig. 4.3) of the curve is computed. This is done by taking the sign of the integral of the difference between the time-series and the initial $y_{f,\infty}$:

$$\hat{s} \cdot S = \text{sgn} \left(\int_{\tau_0}^{\tau_{\max}} (y_f(\tau) - y_{f,\infty}) d\tau \right)$$

where τ_0 is a value in the beginning of the time-series, and τ_{\max} is the value at the end of the selected interval. If integral is negative, the shape is inverted by multiplying both the input data and $y_{f,\infty}$ by -1 , which is equivalent to multiply y_f by $\hat{s} \cdot S$ in any case. To avoid introducing a new variant of y , assume $\hat{s} \cdot S > 0$. The initial guess for A is taken as the difference between the largest value in the possibly inverted time-series and the corresponding $y_{f,\infty}$. The parameter k is estimated through the derivative

$$Ca \exp(a\tau) = \frac{d}{d\tau} y \approx \frac{y_f(\tau + \Delta\tau) - y_f(\tau)}{\Delta\tau}$$

When $\tau = 0$

$$Ca \approx \frac{y_f(\Delta\tau) - y_f(0)}{\Delta\tau} \approx -\frac{C}{\Delta\tau}$$

Therefore

$$a \approx -\frac{1}{\Delta\tau}$$

which is used as the initial guess for a .

Some time-series may converge very rapidly. In this case, $\text{Pr}(y)$ value computed from the procedure outlined above, may be too low. Therefore, the curve fitting is repeated on a smaller part of the data filtered data, and the largest $\text{Pr}(y)$ is reported. Due to the stability problems mentioned above, the mean value of y over the end of the signal is used instead of the computed $y_{f,\infty}$

4.5.2 Finding quantity distribution among particles

The distribution of a quantity among particles is found by computing the corresponding histogram, normalised in the sense that the sum of all bin values equals one. The number of bins in the histogram is chosen according to the Freedman and Diaconis rule so that the number of bins n are determined by

$$n = \frac{N^{1/3}}{2 \text{IQR}(x)} \left[\max_{k \in [0, N]} (y_k) - \min_{k \in [0, N]} (y_k) \right] \quad (4.15)$$

where $\text{IQR}(x)$ is the interquartile range of x .

4.5.3 Determining the fractal dimension

To bootstrap the non-linear solver for the fractal dimension, both sides in eq. (4.2) are transformed using a logarithm of both sides. Then, the model is converted to linear model

$$V_k = \kappa R_k^\beta \iff \log(V_k) = \log(\kappa) + \beta \log(R_k)$$

This expression is then used with a linear least squares solver to bootstrap the non-linear solver. The fractal dimension varies between different iterations and thus, its value is assigned according to the procedure described in section 4.5.1.

4.5.4 Determining particle size distribution

The particle size distribution is found by creating a histogram whose number of bins is determined by eq. (4.15). Instead of fitting the histogram data to the probability density function given by eq. (4.5), the cumulative density function is used. This avoids potential problems with very rare events which would have a zero probability due to the finite number of events. From eq. (4.5)

$$1 - F\left(\frac{R_{\max}}{r_{\max}}\right) = 1 - \left(1 - \exp\left(\gamma \frac{R_{\max}}{r_{\max}}\right)\right) = \exp\left(\gamma \frac{R_{\max}}{r_{\max}}\right)$$

where F is the cumulative distribution function. The function F is measured by computing the cumulative sums of the histogram. This adds more stability, since summation tends to reduce noise.

In order to find γ , bootstrapping is needed. This procedure is similar to the procedure used for the exponential model in section 4.5.1 is used. The only differences here are that no filter is used, and that the shape is known a priori. Also, the current model has no offset.

5 Results

When evaluating the results, it must be remembered that the models are stochastic, and therefore all quantities are compared between two independent runs, but with the same set of parameters. The data from these two runs are then plotted against each other in a correlation diagram as in fig. 5.6. If the quantity is random, then the sampled values will not end up along the identity line. If they do not, the quantity may still be deterministic in the limit, but there are too little data to draw any conclusions.

5.1 The relation between particle volume and radius

When varying the event rates as described in section 4.4.1, the fractal dimension β , defined by eq. (4.2) and retrieved from simulations, vary between 1.5 and 2.2, for both particles in the cloud, and for dropped particles. With $\text{Pr}(\beta) > 1$ (see section 4.5.1), the results also indicate that β converge. For a table concluding these results, see table F.1.

One way to determine if β is affected by any event rate, is to plot it as a function of each of the varied parameters as in fig. 5.1. This figure shows that the fractal dimension decreases when R_d increases. There is also a small decrease in β when R_m increases, and a small increase in β when R_g increases. Another effect is that increasing R_g makes the range of computed values of β smaller. The opposite holds when increasing R_m or R_d . The correlation diagrams of β obtained from two identical runs with different seed are shown in fig. 5.2. The figure indicates all data points gather along the line $y = x$ and therefore β is well defined.

The same kind of diagrams for the normalised volume growth coefficient α are shown in figs. 5.3 and 5.4. The correlation diagrams in fig. 5.4 show a rather weak correlation, especially among the cloud particles. The correlation between different runs among dropped particles is somewhat higher, but not as high as the correlation for β between different runs. This figure also shows that the value of α tends to stay around 1 given the tested parameters. The results are concluded table F.2. In all cases, and thus the quantity converged in most of the cases.

The relation between β and the geometry parameter a is shown in fig. 5.5. This figure indicates that a smaller a , or equivalently longer crystal prototypes, gives a slightly higher β . The corresponding correlation diagrams are shown in fig. 5.6. In this case, the correlation diagrams indicate that the correlation between two different

5 Results

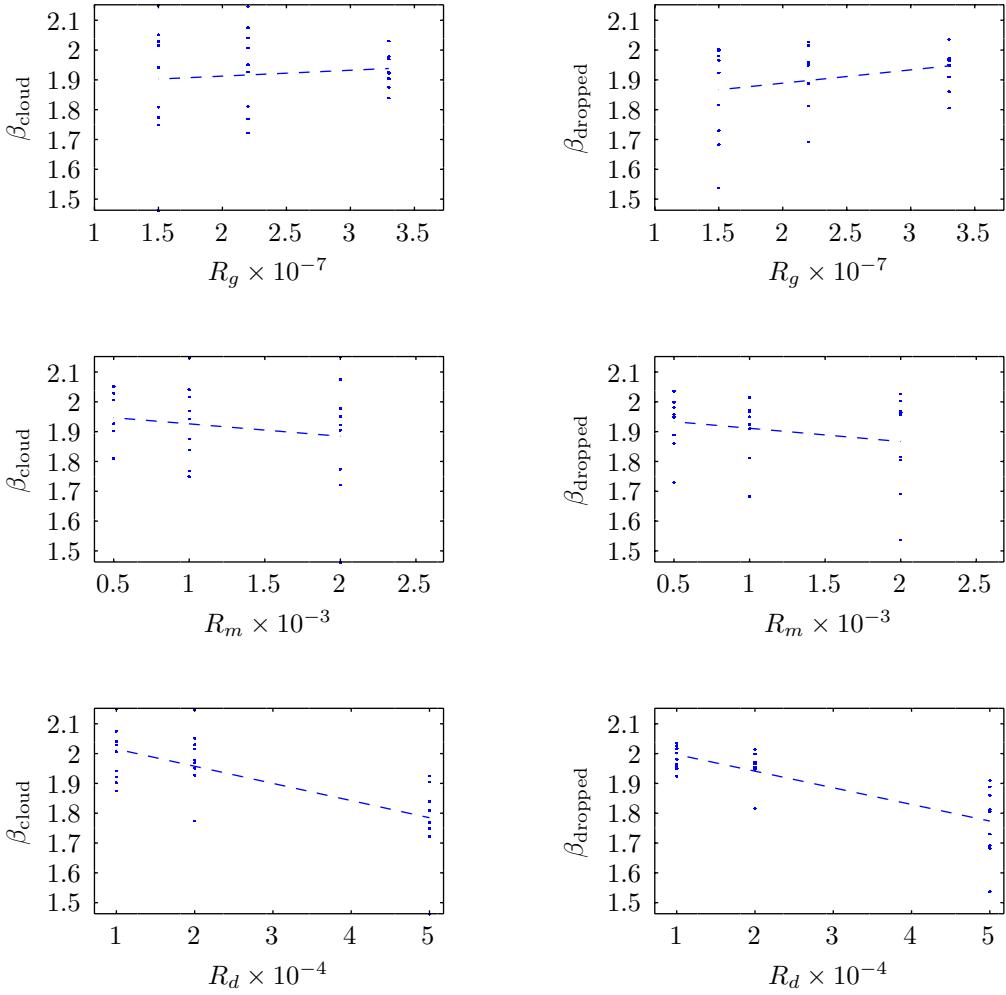


Figure 5.1: The fractal dimension β of particles generated by model B, as a function of different event rates. The values were computed following the procedure described in section 4.5.1 with parameter values listed in table 4.2.

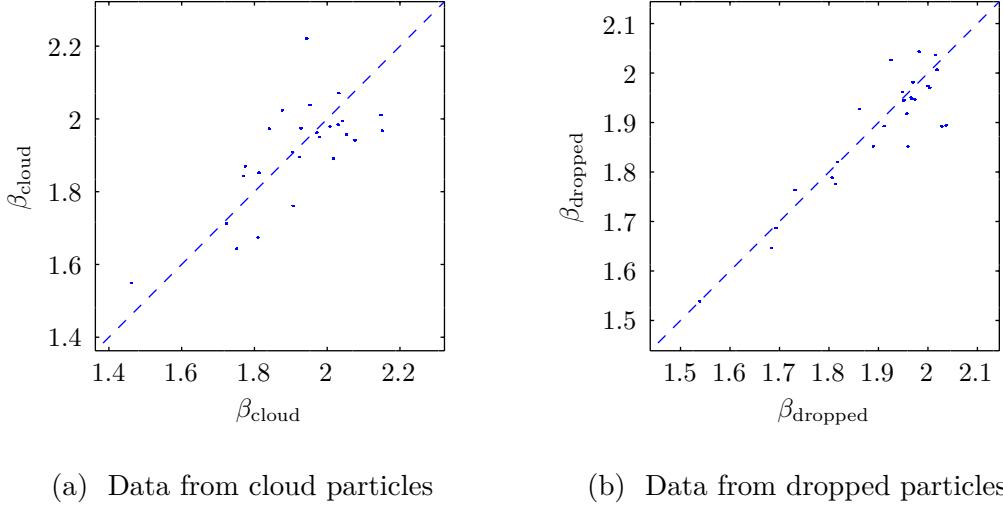


Figure 5.2: Correlation diagrams between the fractal dimension β computed from two different runs of model B with different seed, but the same parameter sweep. The dashed line marks the identity line $y = x$. The values were collected in the same way as for fig. 5.1.

runs is weak, or there are too little data to draw any conclusions. Therefore, it is not possible to claim that there is any relation between a and β . The results for one run are listed in table F.3. For all estimates of β , $\text{Pr}(\beta) > 1$, again indicating convergence.

While α seems to be unaffected by the choice of any event rate, it is affected by the geometry parameter a . Figure 5.7 shows that α has a maximum around $a = 0.45$, and that α varies between 0 and 1.4. The correlation diagrams for these simulations are shown in fig. 5.8. These diagrams show a much higher correlation than fig. 5.4. The results are concluded table F.4. In all cases $\text{Pr}(\alpha) > 1$, and thus the quantity converged in most of the cases.

Figures 5.1 to 5.8 do not say anything about how well the power-law from eq. (4.2) fits with the collected data. To get an idea of how well the model works, V as a function of R_{\max} is shown in fig. 5.9, for a particular set of parameters. From this figure, the model seems to produce data that fits with a power-law, with exception for particles that are built of only one crystal prototype. The power law is also confirmed by fig. 5.16, which contains all simulation data.

As mentioned in section 4.1, β is expected to be equal to 2. Figures 5.1, 5.2, 5.5 and 5.6 show that its value tends to be a bit lower in general, but also that there are parameters that leads to $\beta > 2$. The normalised volume growth coefficient α is expected to be equal to 0.96. From fig. 5.7, this can be achieved by an appropriate choice of geometry parameter a .

5 Results

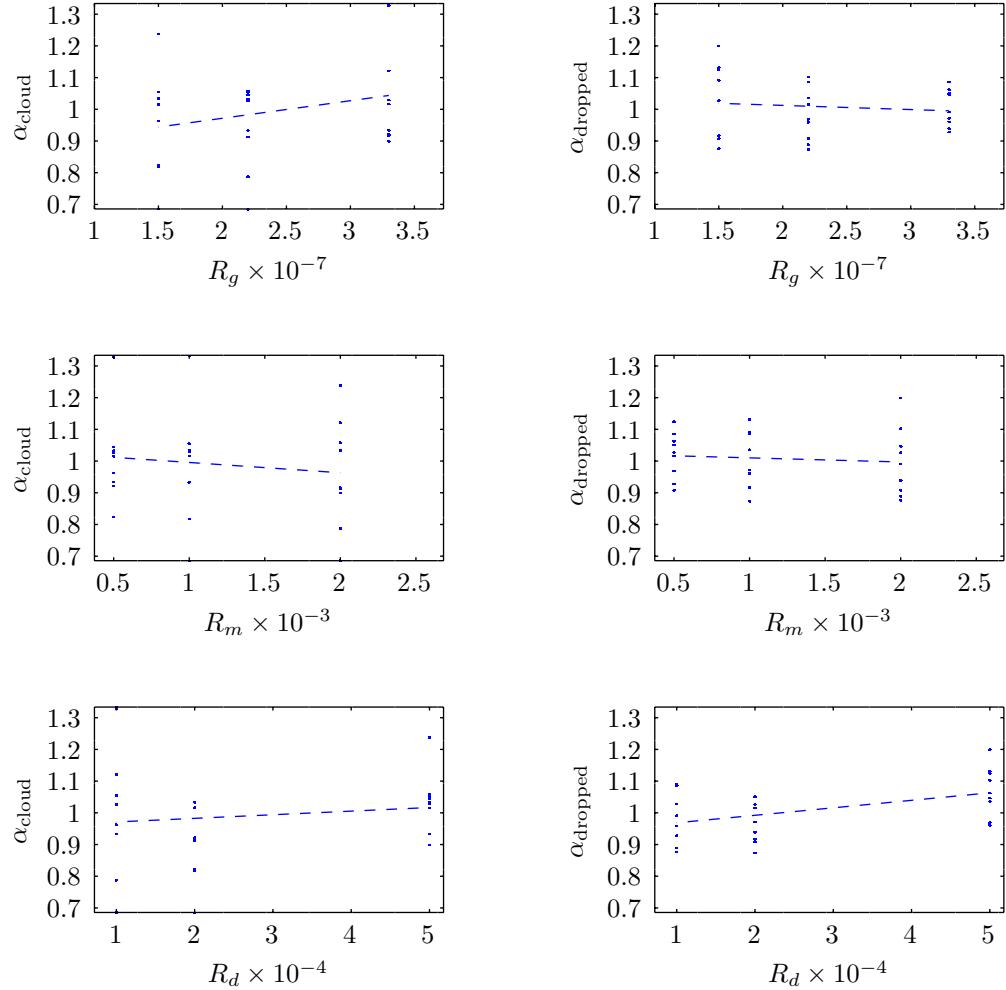


Figure 5.3: The normalised volume growth coefficient α of particles generated by model B, as a function of different event rates. The values were computed following the procedure described in section 4.5.1 with parameter values listed in table 4.2.

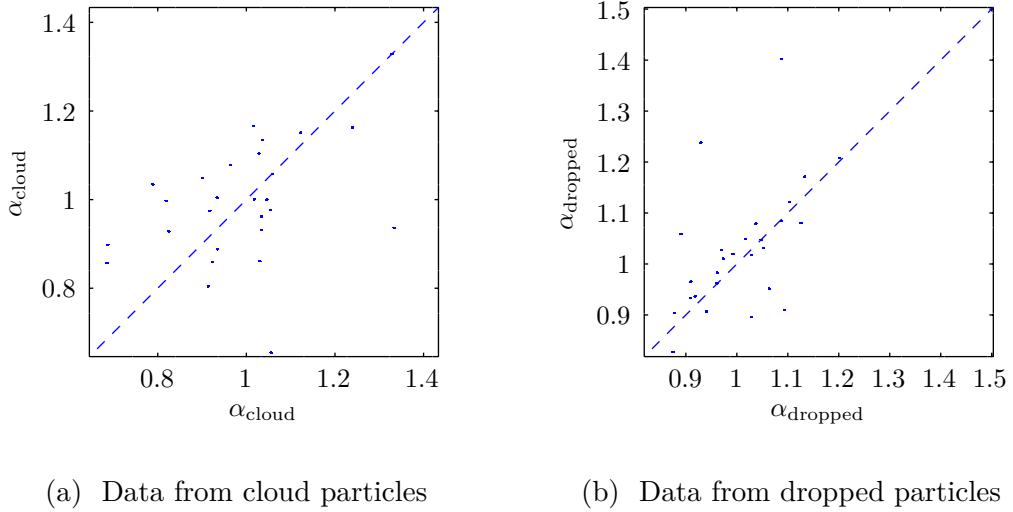


Figure 5.4: Correlation diagrams between the normalised volume growth coefficient α computed from two different runs of model B with different seed, but the same parameter sweep. The dashed line marks the identity line $y = x$. The values were collected in the same way as for fig. 5.3.

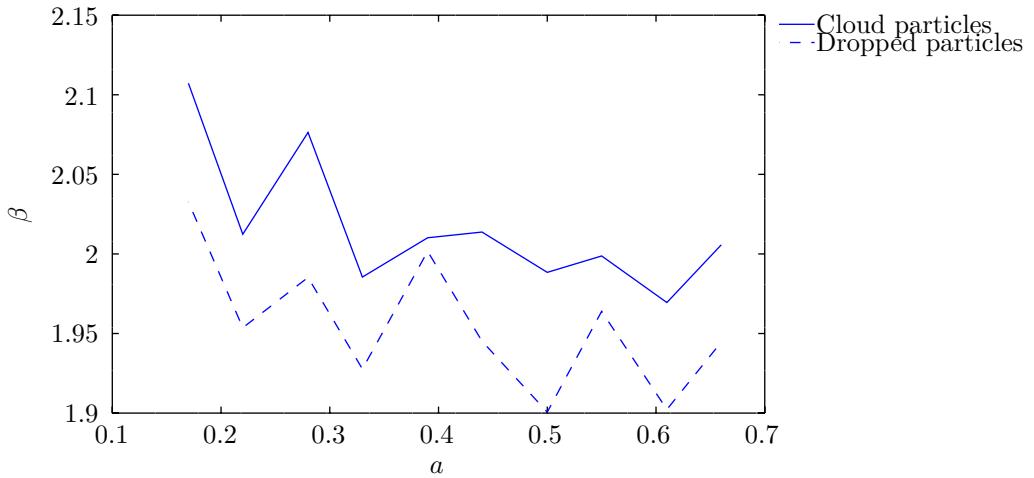


Figure 5.5: Relation between crystal prototype length and the fractal dimension β for particles generated by model B. The solid line is for particles in the cloud, and the dashed line is for dropped particles. The values were computed following the procedure described in section 4.5.1 with parameter values listed in table 4.3. The axial ratio can be found from eq. (4.11).

5 Results

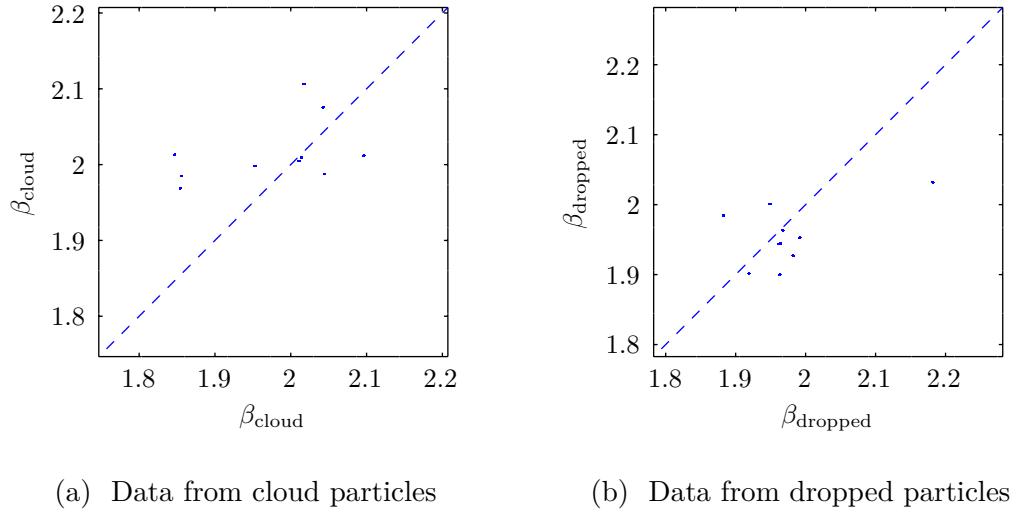


Figure 5.6: Correlation diagrams between the fractal dimension β computed from two different runs of model B with different seed, but the same parameter sweep. The dashed line marks the identity line $y = x$. The values were collected in the same way as for fig. 5.5.

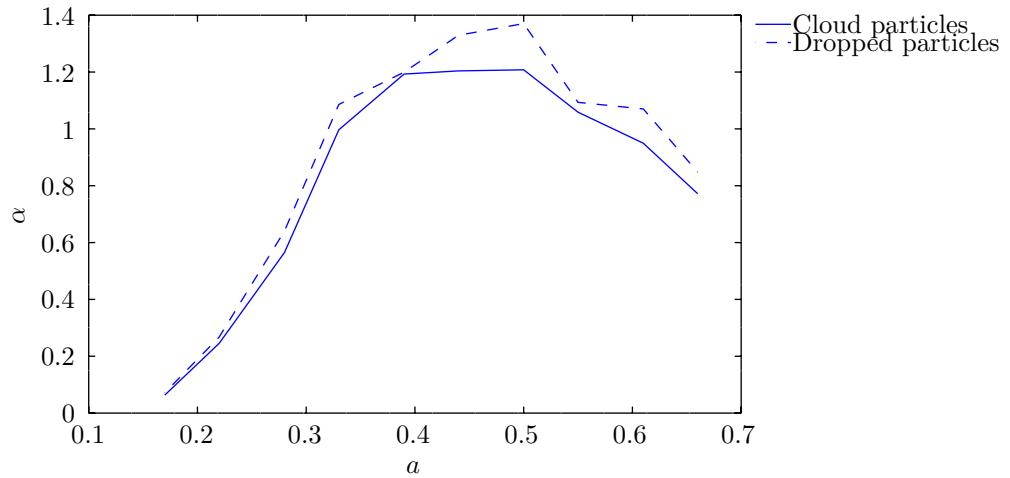


Figure 5.7: Relation between crystal prototype length and the normalised volume growth coefficient α for particles generated by model B. The solid line is for particles in the cloud, and the dashed line is for dropped particles. The values were computed following the procedure described in section 4.5.1 with parameter values listed in table 4.3. The axial ratio can be found from eq. (4.11).

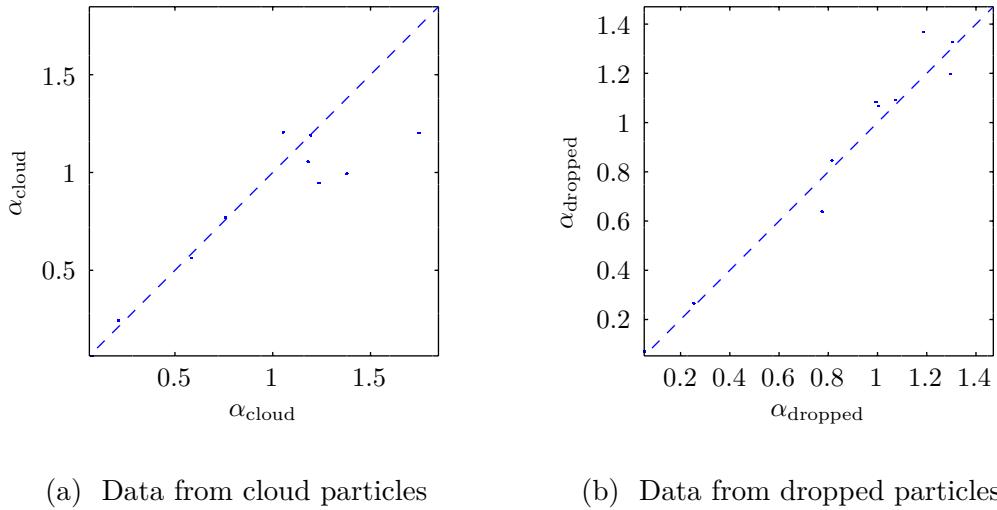


Figure 5.8: Correlation diagrams between the normalised volume growth coefficient α computed from two different runs of model B with different seed, but the same parameter sweep. The dashed line marks the identity line $y = x$. The values were collected in the same way as for fig. 5.7.

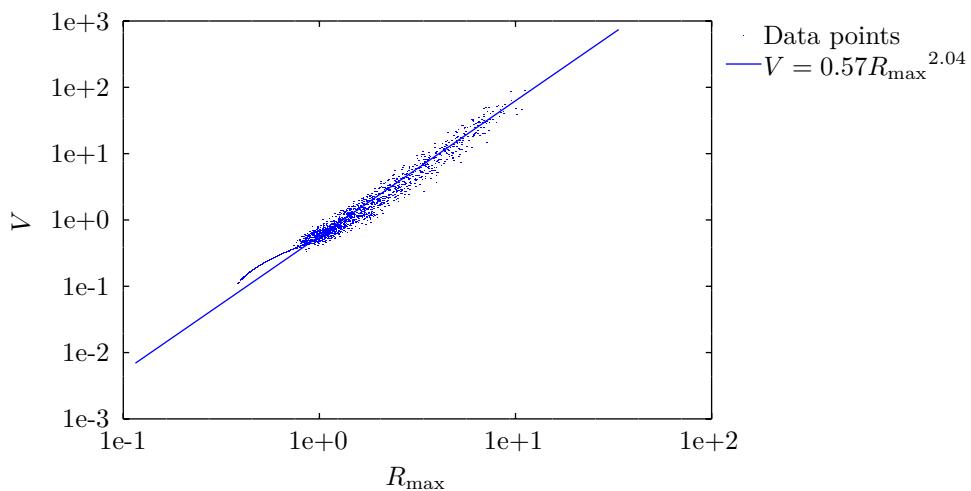


Figure 5.9: Particle volume V as a function of R_{\max} in system units, for particles generated by model B. This result was obtained with $R_g = 33 \times 10^6$, $R_m = 500$ and $R_d = 10\,000$. Other parameters are the same as for fig. 5.1.

5.2 The size distribution parameter

The simulations with different event rates gives γ between -2 and -0.5 . Since $\text{Pr}(\gamma) > 1$, the values seem to converge. The results are listed in table F.5. From fig. 5.10, the size distribution parameter among particles inside the cloud γ_{cloud} , increases with R_g , and decreases with R_m and R_d . This means that a higher R_g , and smaller R_d and R_g implies larger particles. The same holds for the size distribution parameter among the dropped particles γ_{dropped} . This conclusion is supported by the correlation diagrams in fig. 5.11, which shows that both γ_{cloud} and γ_{dropped} between two different runs with the same parameter sweep are correlated.

The relation between γ and the geometry parameter a is shown in fig. 5.12. This figure shows that γ is largest when $a = 0.44$. The more extreme ratio between a and L , the smaller are the generated ice particles. The corresponding correlation diagrams are shown in fig. 5.13. For a table concluding the results, see table F.6. Also in this case, $\text{Pr}(\gamma) > 1$.

To see whether or not eq. (4.5) correctly describes the particle size distribution, a particular distribution is plotted in fig. 5.14. This figure shows that the model fits pretty well in slope, with exception for the apparent high frequency of large particles in the simulation output, but since the creation of larger particles is rare, there are much more uncertainties in that end of the figure.

Compared to real measurements referenced in section 4.2, all data-sets analysed so far corresponds to the smaller or equivalently, less complex, snowflakes found in measurements. It is possible to create larger particles by increasing the lifetime of particles inside the cloud. For example, the choice $a = 0.33$, $R_d = 500$, $R_m = 2000$, and $R_g = 44 \times 10^6$ results in $\gamma_{\text{cloud}} = -0.0949$ and $\gamma_{\text{dropped}} = -0.115$. The resulting particle size distribution from one such run is shown in fig. 5.15. In this case $\beta_{\text{cloud}} = 1.88$, $\alpha_{\text{cloud}} = 2.90$, $\beta_{\text{dropped}} = 1.87$, and $\alpha_{\text{dropped}} = 2.69$.

If r_{\max} is expected to be unaffected by particle size, Matrosov, S. Y. (2007) suggests that larger particles should result in a larger value of α . The results indicates that this is the case. As seen from fig. 5.7, it is possible to shrink α by using more extreme aspect ratios for the crystal prototypes. With the same parameters as above, but with $a = 0.22$, gives $\gamma_{\text{cloud}} = -0.138$, $\gamma_{\text{dropped}} = -0.185$, $\alpha_{\text{cloud}} = 0.64$, $\beta_{\text{cloud}} = 1.92$, $\alpha_{\text{dropped}} = 0.26$, and $\beta_{\text{cloud}} = 2.18$.

5.3 The spherical volume fill ratio

From fig. 5.16, the trend for the average spherical volume fill ratio v_{r0} defined by eq. (4.7) is that it decreases with slower growth rate R_g . From the same figure, increasing the melt rate R_m or drop rate R_d , increases v_{r0} . The correlation diagrams in fig. 5.17, indicates that there is a true correlation. The underlying values are listed in table F.7. For all tested parameter, $\text{Pr}(v_{r0}) > 1$, indicating convergence.

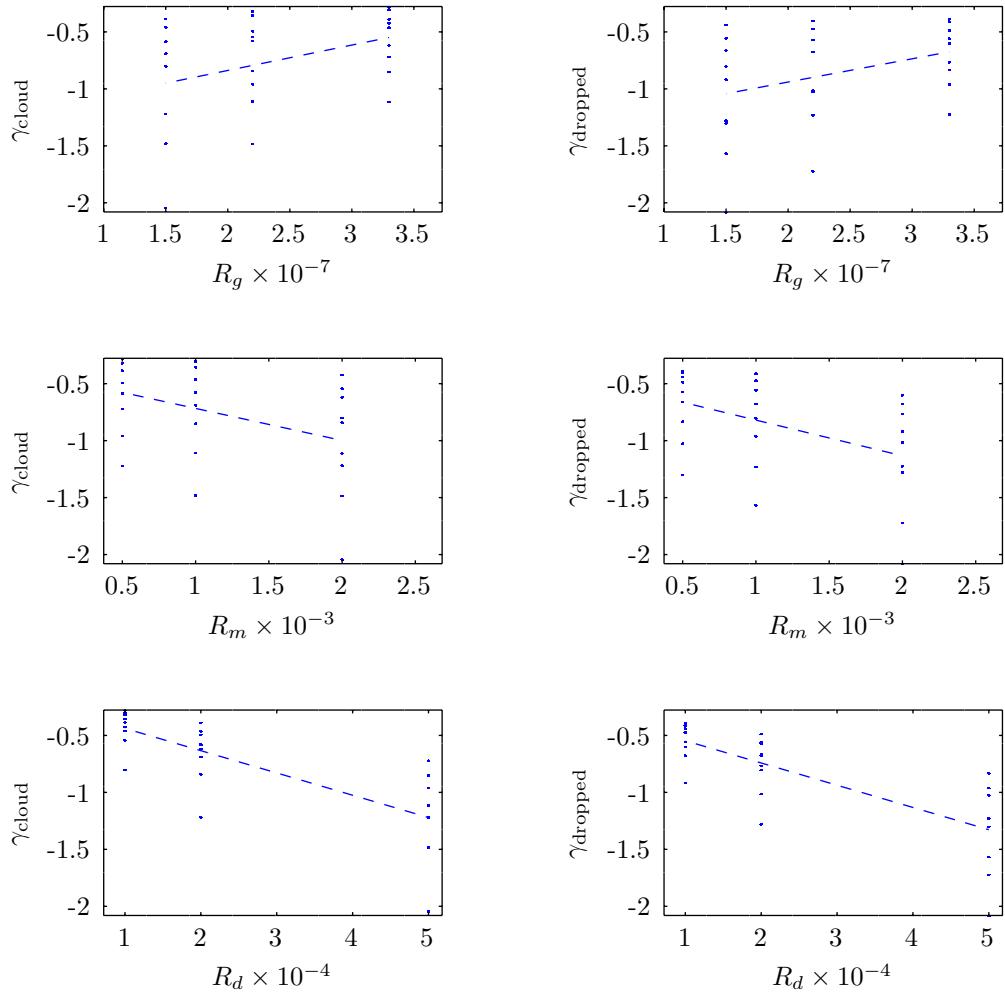


Figure 5.10: The size distribution parameter γ among particles generated by model B, as a function of different event rates. The values were computed following the procedure described in section 4.5.1, with the parameter values listed in table 4.2.

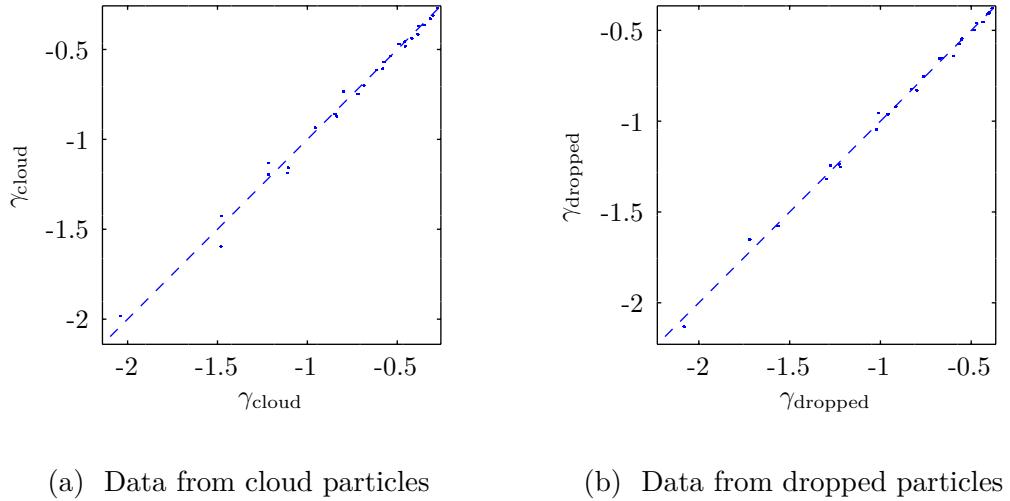


Figure 5.11: Correlation diagrams between the size distribution parameter computed from two different runs of model B with the same parameter sweep. The dashed line marks the identity line $y = x$. The parameter values used are those found in table 4.2. The values were collected in the same way as for fig. 5.10.

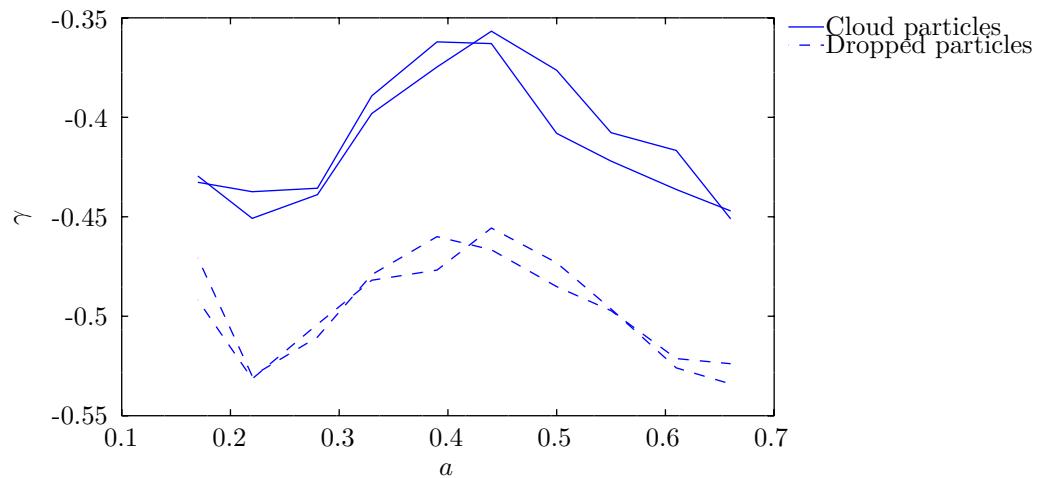


Figure 5.12: Relation between crystal prototype length and the size distribution parameter γ among particles generated by model B. The solid line is for particles in the cloud, and the dashed line is for dropped particles. The values were computed following the procedure described in section 4.5.1, with the parameter values listed in table 4.3. The axial ratio can be found from eq. (4.11).

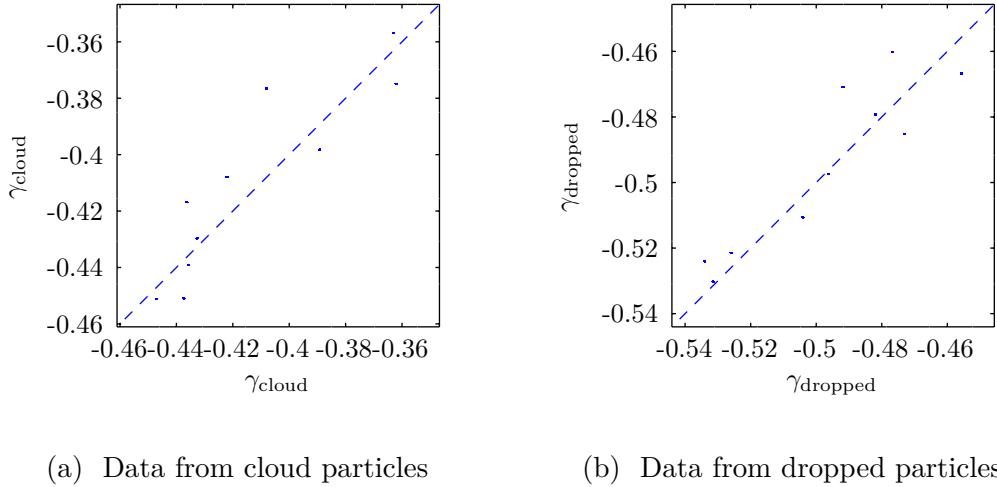


Figure 5.13: Correlation diagrams between the size distribution parameter computed from two different runs of model B with the same parameter sweep. The dashed line marks the identity line $y = x$. The values were collected in the same way as for fig. 5.12.

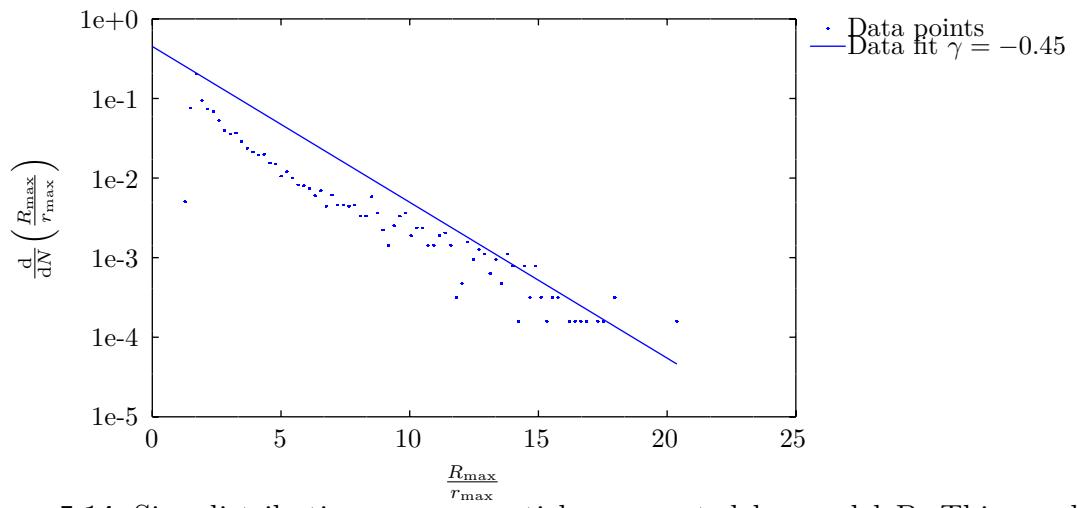


Figure 5.14: Size distribution among particles generated by model B. This result was obtained with $a = 0.44$. Other parameters are the same as for fig. 5.13.

5 Results

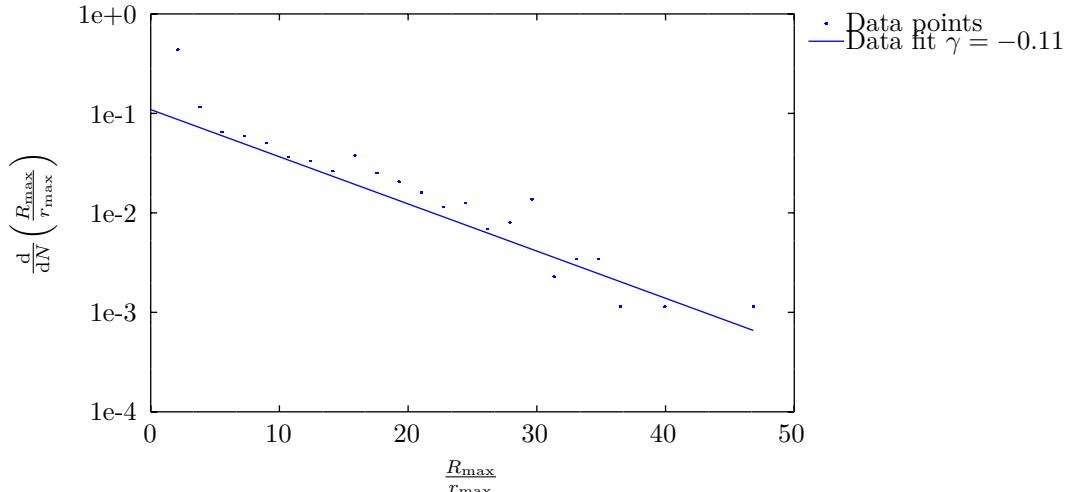


Figure 5.15: Size distribution among dropped particles generated by model B. This result was obtained with $R_g = 44\,000\,000$, $R_m = 2000$ and $R_d = 500$. Other parameters are the same as for fig. 5.11.

The effect on v_{r0} of making the aspect ratio of the crystal prototypes more extreme is that its value decreases, as seen in fig. 5.18. As seen in fig. 5.19, the correlation is strong also in this case. The underlying values are listed in table F.8. For all tested parameter, $\text{Pr}(v_{r0}) > 1$, indicating convergence.

To see the relation between R_{\max} and v_r , the ratio $\frac{v_r}{v_{r0}}$ from eq. (4.9) is plotted. The result is shown in fig. 5.20. By multiplying the values in this plot with v_{r0} computed from different parameters, the spherical volume fill ratio is obtained. Figure 5.20 also compares the ratio obtained from all simulations with the theoretical expression eq. (4.10) derived in section 4.3. The data points seem to end up parallel to the theoretical line, but somewhat shifted.

From figs. 5.16 and 5.18, the pseudo-average of the spherical volume fill ratio varies between 0.02 and 0.1 for particles that are not generated from very thin crystal prototypes. For particles generated from the thinnest crystal prototypes ($a = 0.17$), the value goes down to 3×10^{-3} . When multiplying these values by values for the reduced spherical volume fill ratio from fig. 5.20, it is found that the spherical volume fill ratio varies between 6×10^{-4} and somewhere around 1, depending on particle size and shape.

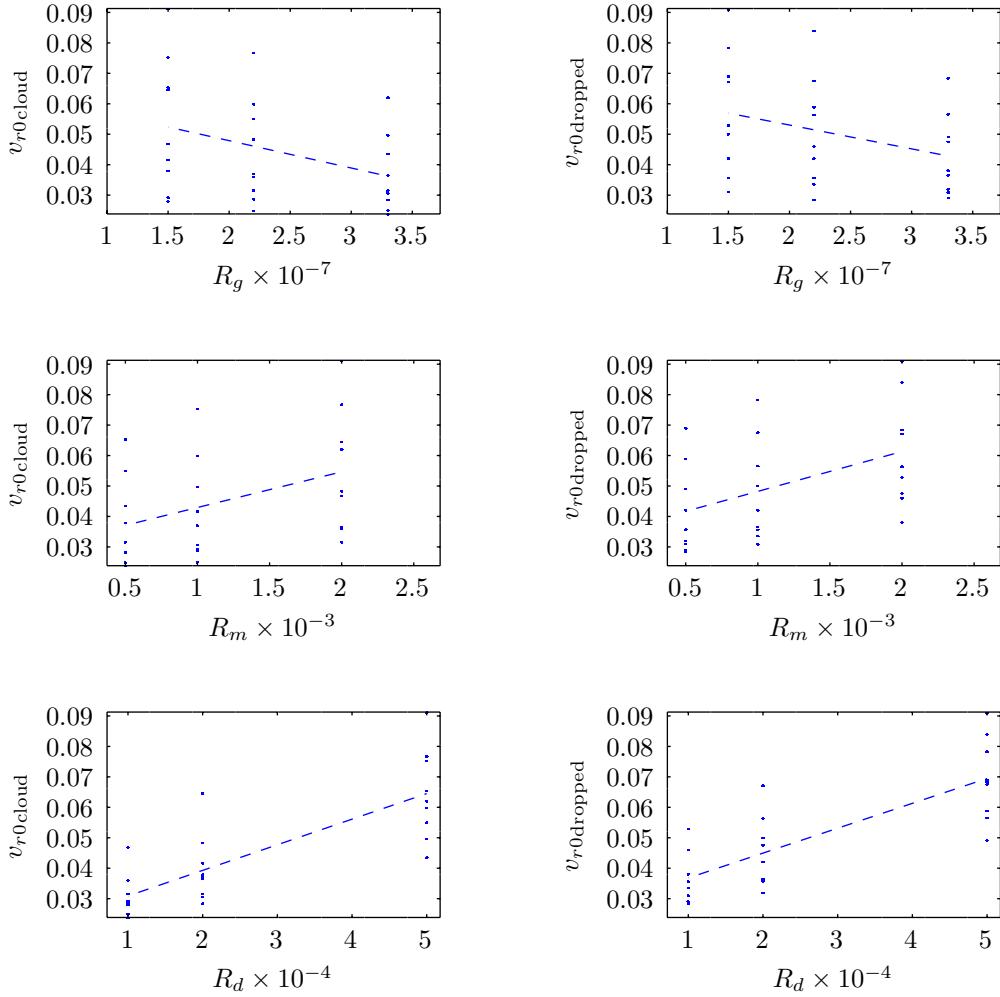


Figure 5.16: The pseudo-average spherical volume fill ratio v_{r0} defined by eq. (4.7), of particles generated by model B, as a function of different event rates. The values were computed following the procedure described in section 4.5.1, with the parameter values listed in table 4.2.

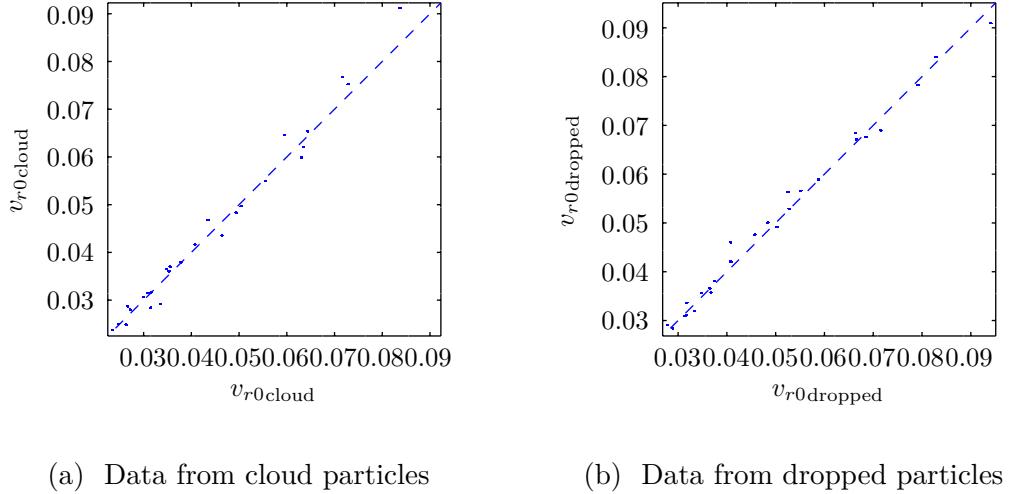


Figure 5.17: Correlation diagrams between the pseudo-average spherical volume fill ratio v_{r0} defined by eq. (4.7), computed from two different runs of model B with different seed, but the same parameter sweep. The dashed line marks the identity line $y = x$. The values were collected in the same way as for fig. 5.16.

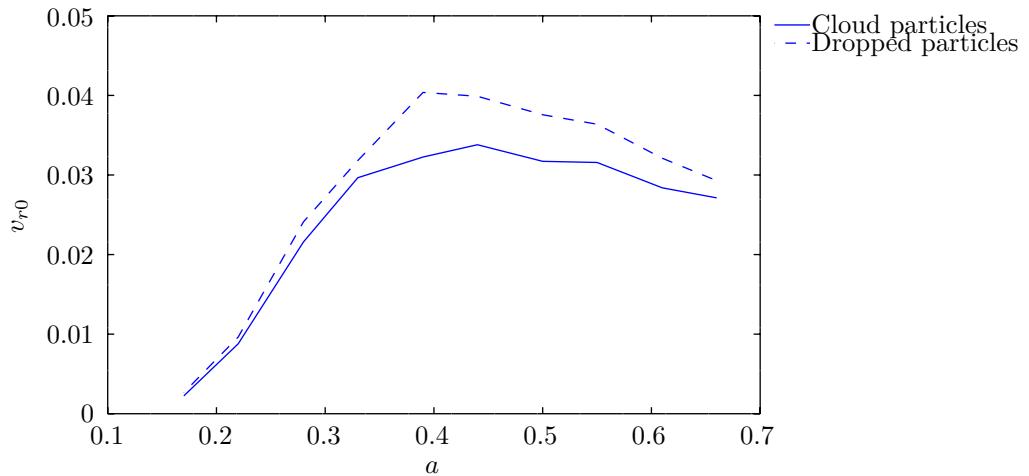


Figure 5.18: Relation between crystal prototype length and the pseudo-average spherical volume fill ratio v_{r0} defined by eq. (4.7), for particles generated by model B. The solid line is for particles in the cloud, and the dashed line is for dropped particles. The values were computed following the procedure described in section 4.5.1, with the parameter values listed in table 4.3.

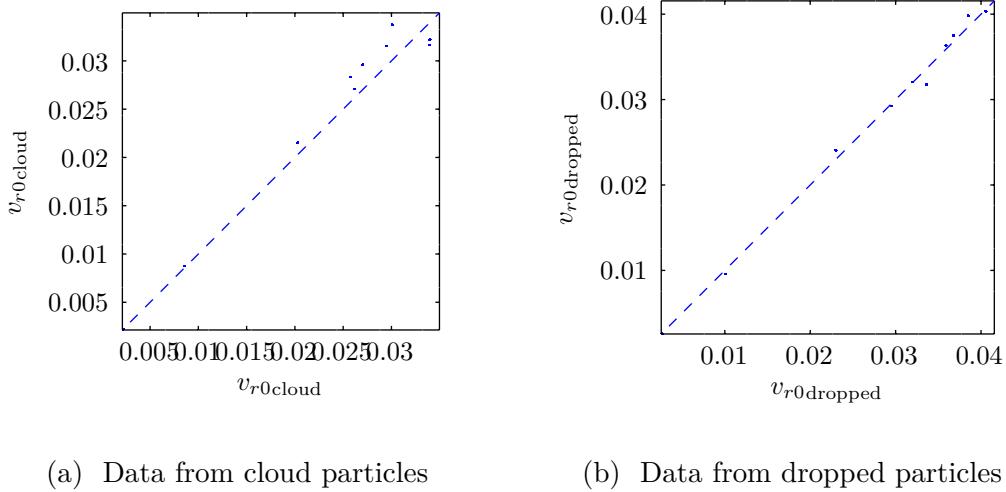


Figure 5.19: Correlation diagrams between the average spherical volume fill ratio v_{r0} defined by eq. (4.7), computed from two different runs of model B with different seed, but the same parameter sweep. The dashed line marks the identity line $y = x$. The values were collected in the same way as for fig. 5.18.

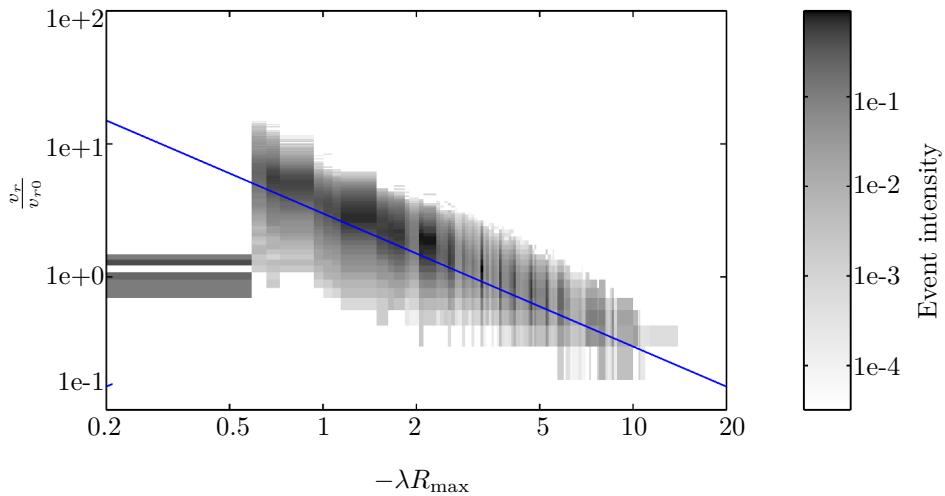


Figure 5.20: The reduced spherical volume fill ratio defined by eq. (4.9), as a function of $-\lambda R_{\max}$. The scatter-plot contains all data points from every simulation. The line shows the relation $\frac{v_r}{v_{r0}} = -\frac{3}{\lambda R_{\max}}$, that is eq. (4.10) derived in section 4.3. The intensity is logarithmic and represents the abundance of events at the specific point weighted by the number of events for the corresponding λ .

6 Discussion

This project has shown that algorithms from the study of computational geometry can be useful when generating snow particle shape data. From the data analysis from the output of model B, it appears that the model can, for large particles, reproduce the volume growth coefficient, fractal dimension, and size distribution of real particles. For the smallest generated particles, both the fractal dimension and size distribution deviate from what is expected, but in this range it is harder to perform measurements accurately due to “shattering on the housing of probes” (Baran, A. J. 2012).

The particle aspect ratio, and speed, has been recorded, but not analysed. This is also true for the system dynamics. There is no real reason for skipping this analysis, other than the fact that such analysis would have bloated the report. The aspect ratios recorded are axis aligned. Another way of measuring the aspect ratio is to use the principal axes, but in its current state, there is no function for computing principal components within the toolkit.

6.1 Ways to improve the toolkit

In its current state, the toolkit provides basic functionality for generating composed particles, and that is probably enough to implement very sophisticated models, but there is room for improvements of the toolkit. The following list shows some points of improvements:

- It would be a benefit if the toolkit could divide imported geometry into convex parts by itself. Then, it would be easier to make new ice crystal prototypes. Also, the particle deformation described in section 2.3.1 can be less restrictive.
- Implement principal component analysis. Besides the possibility to measure aspect ratios relative to the principal axes, it makes it possible to implement a model that takes the normal to the largest spanned parallelogram as falling direction.
- Use composed transformations in a larger extent in order to save computation time
- If it can be shown that the most realistic shapes are generated from crystal prototypes with a small number of vertices, a limit can be added to the

number vertices a subvolume may contain. Such a limit may reduce the number of memory cache misses due to increased memory locality. This comes from the fact that a variable number of vertices requires an additional pointer dereference.

6.2 Extensions and correctness of the models

Even though it is straight forward to implement anisotropy to the coalescence process, this has not been done within the project. The reason for this is that more simulations adds more results, and the project has already resulted in more data than could be analysed.

There are some cheating in computing the matrix elements in model B namely that events that would result in an overlap (see fig. 3.2), should have a probability of zero so they never occur. It is possible to implement this, but then the probability matrix needs to store the probabilities on face level, rather than on particle level, and this would result in a much larger matrix. Also, in order to compute the matrix element, collision detection has to be performed between all faces, which certainly is impractical.

It could be interesting to add the equilibrium between solid and liquid water described by eq. (3.7). Then coalescence of water droplets, as well as water droplets and ice crystals, also have to go into the model. Adding water droplets does not add much new to the algorithm described in section 3.4.3, since water droplets can be thought of as a spheroid crystal prototype, but the coalesce process will differ for the coalescence of water droplets, if they are not assumed to freeze directly. Another difference between water droplets and ice particles lie in their density. Therefore, the particle density has to be considered when computing the terminal velocity.

When running simulations, the crystal prototype radius has been kept constant. It is not difficult to modify the code so this quantity can be randomised as well. Varying both length and radius may lead to a larger fractal dimension, since this gives a two-dimensional scaling, instead of a one-dimensional scaling.

6.3 Alternative ways of analyse data

Besides testing other models, natural ways of continuing the work within this field include studying more aspects of the evaluated model such as the distribution of particle aspect ratios with respect to their principal component axes, and the dynamics of the simulated system.

Instead of using R_{\max} , one can measure D_{\max} and the latter measurement is actually more easy to compute. However, since R_{\max} was used for estimating cross-sections, it was used in data analysis as well. Because R_{\max} is based on the location

of the centroid, there can be some differences in particle sizes. It is unlikely to affect the fractal dimension, but it may affect the volume growth coefficient, since it may happen that R_{\max} is no longer exactly equal to $\frac{D_{\max}}{2}$.

Another way of analysing data is to compare some of the generated particles with photographic data of real snowflakes. Indeed, since the framework contains functionality needed to communicate with any tool that accepts geometry data in the **Wavefront** file format (FileFormat.Info 2015), it is straightforward to feed the simulation output into either a scanline rasteriser or a ray-tracer.

A scanline rasteriser can be used to compare the geometrical features in the output of the simulation with real data. While there are analysis techniques for such images (Schmitt, C. G. et al. 2014), this kind of analysis has not been done during this project.

A ray-tracer has been used to generate the cover picture. A ray-tracer makes it possible to analyse optical properties as well as geometrical features, but then the background of the real data should be the same as the background and lightning used when running the ray-tracer. A good background might be a checker-board pattern in the scale of individual ice crystal prototypes, since it reveals the refraction effect. One should also keep in mind that a ray-tracer is at the time of writing very slow compared to a scanline rasteriser, and therefore, ray-tracing is only beneficial if there are models of weather or climate, that in some way affects or is affected by, the index of refraction within the optical region.

6.4 Uncertainties in data

As mentioned in section 4.1, the value used for the size normalising constant r_{\max} was obtained by an estimate from one image of a real ice particle, which makes this value not very accurate. This uncertainty mainly affects the complexity of particles: a smaller r_{\max} requires a more complex aggregate given the same aggregate size, and vice versa. Assuming this measurement indeed is correct, the distance was measured orthogonal to the particle length, which introduces another uncertainty since the latter is unknown. That is because the real length particle size is $r_{\max} = \sqrt{L^2 + r_{\perp}^2}$, where L is the length of the crystal prototype, and r_{\perp} is the radius measured in the picture. One possible way to deal with the latter problem is to run simulations and see how L relates to other measurable quantities. It may then be possible to set up a self-consistency condition, which gives the particle aspect ratio.

7 Conclusions

During the project, a toolkit that can be used to generate snow particles, and rasterise them, has been developed. The toolkit was used to implement a Gillespie based method that was evaluated by letting it generate snow particles. By using the data analysis methods described in chapter 4, it was found that all of the studied time-series converged, and that the algorithm generated particles, that exhibit some properties also found among snow particles in nature. Additionally, it was found that

- the fractal dimensions of the generated particles appear to be quite insensitive to the prototype aspect ratio, but there is a tendency for their fractal dimensions to grow with the number of particles within the cloud
- the growth coefficient is mainly affected by the prototype aspect ratio, and not by the event rates. The growth coefficient is largest for cubic particles, and decreases when the prototype aspect ratio becomes more extreme.
- the spherical volume fill ratio is both affected by the prototype aspect ratio, and the event rates. With respect to the prototype aspect ratio, the spherical volume fill ratio behaves similar to the growth coefficient. The spherical volume fill ratio decreases with the number of particles within the cloud.
- to generate large particles, the particle lifetime within the cloud should be long
- for tested parameter, particles within the cloud tends to be larger than dropped ones

During the construction of analysis methods, it was found that it is possible to construct a relation between volume fill ratio, averaged volume fill ratio, and the particle size, given the size distribution parameter. Although the expression was derived assuming an exponential distribution among the particle sizes, it should be possible to derive that kind of expressions for different size distributions. The benefit of such a relation is that it is easier to compare different data sets retrieved by simulations or observations, provided that both sets follow the same size distribution.

Glossary

Advanced Vector Extensions An extension to the x86 architecture that allows hardware support for arithmetic on multiple numbers. 28

API Application Programming Interface. 4

Application Programming Interface A set of routines, protocols, and tools for building software and applications. 4

AVX Advanced Vector Extensions. 28

bootstrapping The procedure of starting a procedure without any additional input that normally would require such input. 38, 42

bounding box The minimal cuboid that completely encloses a body. 11, 36

Cloud Particle Imager A device that can be used to capture images of cloud particles. 32

convex set A set such that for all points P_l on the line segment between any two points in the set, P_l belongs to the set. 7, 9, 11, 13

CPI Cloud Particle Imager. 32

crystal prototype Template geometry used to initialise ice particle generation. 4, 5, 9, 29, 31, 35, 43, 45, 47, 48, 50, 52, 54, 56, 59

DDA Discrete Dipole Approximation. 2, 3, 5, 6, 17

depth first An algorithm for traversing a graph. In this algorithm, the grandchildren takes precedence over siblings. 10

Discrete Dipole Approximation A technique used simulate electromagnetic scattering. 2

edge The link between two nodes in a graph. 7, 8, 10

forward model A model that describes how the interesting quantities are mapped to measured quantities. 1

fractal dimension The exponent in the mass-radius relation. It can be interpreted as the effective number of directions in which the particles grow.. 31, 39, 44

graph A set of nodes connected through edges. Objects that can be represented by a graph includes electrical circuits, tram charts, and road networks. 4, 5, 7, 8, 10, 17

inversion of control A software design in which custom-written portions of a computer program receive the flow of control from a generic, reusable library. 4

MASC Multi-Angle Snowflake Camera. 32

Mie scattering Scattering of electromagnetic radiation with a wavelength that is of the same size as the scattering particles. 1

Multi-Angle Snowflake Camera A device that automatically photographs hydrometeors in free fall. 32

normalised volume growth coefficient A dimensionless version of the volume growth coefficient. 32, 43, 45

polygon mesh A storage model that is frequently used in 3D computer graphics to represent 3D objects. 4–7, 17, 20

POSIX time The number of seconds since 1970-01-01 00:00. 36

prediction ratio The prediction ratio between two models is the ratio between their prediction error. 40

profiling The process of measuring the time spent in different parts of computer program. 27

reduced spherical volume fill ratio The spherical volume fill ratio divided by the pseudo-average of the spherical volume fill ratio. 34, 57

ringing artefacts Initial oscillations in a dynamical system. 40

stack A data structure that stores elements following the last-in first-out scheme. 10, 11

sub-volume A convex subset of an **IceParticle**. 7, 9, 11, 13, 14, 21, 22

system unit The unit of length used for measuring the size of particles.. 4, 35, 49

volume growth coefficient Coefficient measuring how much the volume of a particle is affected by the particle size. 31

voxel A three-dimensional pixel. 11–14

Bibliography

- Baran, A. J. (2012) From the single-scattering properties of ice crystals to climate prediction: A way forward. *Atmospheric Research*, vol. 112, pp. 45–69
doi: [10.1016/j.atmosres.2012.04.010](https://doi.org/10.1016/j.atmosres.2012.04.010)
- Braham, R. R. (1990) Snow Particle Size Spectra in Lake Effect Snows. *Journal of Applied Meteorology*, vol. 29, issue 3, pp. 200–207
doi: [10.1175/1520-0450\(1990\)029<0200:SPSSIL>2.0.CO;2](https://doi.org/10.1175/1520-0450(1990)029<0200:SPSSIL>2.0.CO;2)
- Pseudo-random number generation. (2015) In *cppreference.com*.
<http://en.cppreference.com>. (2015-08-16)
- std::pair. (2015) In *cppreference.com*.
<http://en.cppreference.com>. (2015-09-09)
- Draine, B. T. and Flatau, P. J. (1994) Discrete-dipole approximation for scattering calculations. *Journal of the Optical Society of America*, vol. 11, issue 4, pp. 1491–1499
doi: [10.1364/JOSAA.11.001491](https://doi.org/10.1364/JOSAA.11.001491)
- Eaton, J. W. et al. (2011) *GNU Octave* v. 3.8.0. GNU.
- Wavefront OBJ: Summary from the Encyclopedia of Graphics File Formats. (2015) In *FileFormat.Info*.
<http://www.fileformat.info>. (2015-07-16)
- Garrett, T. J. et al. (2015) Orientations and aspect ratios of falling snow. *Geophysical Research Letters*, vol. 42, issue 11, pp. 4617–4622
doi: [10.1002/2015GL064040](https://doi.org/10.1002/2015GL064040)
- Gillespie, D. T. (1975) An Exact Method for Numerically Simulating the Stochastic Coalescence Process in a Cloud. *Journal of the Atmospheric Sciences*, vol. 32, issue 10, pp. 1977–1989
doi: [10.1175/1520-0469\(1975\)032<1977:AEMFNS>2.0.CO;2](https://doi.org/10.1175/1520-0469(1975)032<1977:AEMFNS>2.0.CO;2)
- Hong, G. (2007) Parameterization of scattering and absorption properties of nonspherical ice crystals at microwave frequencies. *Journal of Geophysical Research*, vol. 112, issue D11208, pp. n/a
doi: [10.1029/2006JD008364](https://doi.org/10.1029/2006JD008364)

BIBLIOGRAPHY

- Intel Corporation (2015) *Intel® 64 and IA-32 Architectures Software Developer's Manual* vol. 1. Intel Corporation.
- Linux Kernel Organization (2013) *Linux Programmer's Manual* vol. 4. Linux Kernel Organization.
- Maruyama, K. and Fujiyoshi, Y. (2005) Monte Carlo Simulation of the Formation of Snowflakes. *Journal of the Atmospheric Sciences*, vol. 64, issue 5, pp. 1529–1544
doi: 10.1175/JAS3416.1
- Matrosov, S. Y. (2007) Modeling Backscatter Properties of Snowfall at Millimeter wavelength. *Journal of Atmospheric Sciences*, vol. 64, issue 5, pp. 1727–1736
doi: 10.1165/JAS3904.1
- Mitchell, D. L., Zhang, R. and Pitter, R. L. (1990) Mass-Dimensional Relationships for Ice Particles and the Influence of Rimming on Snowfall Rates. *Journal of Applied Meteorology*, vol. 29, issue 2, pp. 153–164
doi: 10.1175/1520-0450(1990)029<0153:MDRFIP>2.0.CO;2
- Möller, T. (1997) A Fast Triangle-Triangle Intersection Test. *Journal of Graphics Tools*, vol. 2, issue 2, pp. 24–30
doi: 10.1080/10867651.1997.10487472
- Newman, M. C. (1993) Regression analysis of log-transformed data: Statistical bias and its correction. *Environmental Toxicology and Chemistry*, vol. 12, issue 6, pp. 1129–1133
doi: 10.1002/etc.5620120618
- Nürnberg, R. (2013) Calculating the volume and centroid of a polyhedron in 3d. Imperial Collage London.
<http://wwwf.imperial.ac.uk/~rn/centroid.pdf>. (2015-07-09)
- Rees, W. G. (2001) *Physical principles of remote sensing*. 2ed. Cambridge: Cambridge University Press.
ISBN: 0-521-66948-0
- Schmitt, C. G. and Heymsfield, A. J. (2014) Observational quantification of the separation of simple and complex atmospheric ice particles. *Geophysical Research Letters*, vol. 41, issue 4, pp. 1301–1307
doi: 10.1002/2013GL058781
- Stein, T. H. M., Westbrook, C. D. and Nicol, J. C. (2015) Fractal geometry of aggregate snowflakes revealed by triple-wavelength radar measurements. *Geophysical Research Letters*, vol. 42, issue 1, pp. 176–183
doi: 10.1002/2014GL062170

A System requirements

The toolkit has been developed and tested on **Ubuntu 14.04**, with the compiler **g++ 5.1.0**. One design principle of the project has been to keep the dependency on third-party libraries as small as possible. Nevertheless, the toolkit requires a GNU system together with the Linux® kernel, and the **GLM** library¹. Also, the toolkit requires

- A compiler that supports C++11 (**g++ 4.8** should work)
- A processor that supports AVX-256 instructions (An Intel® IvyBridge or later should work)

To avoid the need for maintaining **make**-files, the **Wand** tool from **Gabi**² version 5.77 has been used. Notice that the toolkit does not have any dependencies to the **Herbs**³ library, and the source code ships with a **wandcfg.spell** file. Therefore, compiling **Gabi** and placing the **wand** executable file in an appropriate directory⁴ should be enough to compile the toolkit. In other words, the formal installation procedure for installing **Gabi** is not needed. That said, the toolkit can be compiled without **Wand** though using **Wand** makes it easier to maintain the **make**-file in case the internal dependency graph of the toolkit changes.

Since the ice crystal prototypes are stored in text files, it is possible to create custom prototypes in a text editor. However, it can be easier to create the shape if WYSIWYG⁵ editing can be used. This is possible by using the 3D modelling and rendering package **Blender**⁶, to draw the geometry for the crystal prototypes, and then using the bundled plug-ins intended for **Blender** to export the geometry into a crystal prototype file.

¹See <http://glm.g-truc.net> (2015-08-11 18:26)

²More information about **Gabi** can be found at <https://github.com/milasudril/gabi> (2015-08-11 18:30). The source code for the stable branch can be found at <https://github.com/milasudril/gabi/releases> (2015-08-11 18:30)

³**Herbs** is the class library shipped with **Gabi**

⁴An appropriate directory is any directory included in the PATH environment variable

⁵What You See Is What You Get

⁶See <https://www.blender.org> (2015-08-11 18:18)

B Retrieving and compiling the toolkit

Below is a shell script that downloads and compiles the toolkit.

```
#!/bin/bash

# Download Gabi
wget -O gabi-5.77.tar.gz "https://github.com/milasudril/gabi/archive/5.77.tar.gz"

# Decompress the tarball
gzip -d "gabi-5.77.tar.gz"

# Extract its contents
tar -xf "gabi-5.77.tar"

# Cd into the Gabi source directory
cd "gabi-5.77/source"

# Compile Gabi
make -f "Makefile-GNULinux64"

# Move Wand to an appropriate place
mv "__wand_targets-x86_64-gnulinux/wand/wand" ~/bin/wand

# Leave the Gabi source directory
cd ../..

# Everything above this line can be skipped if the correct version
# of Wand is already installed
#####
# Download the toolkit (the latest version)
git clone "https://github.com/milasudril/snowflakemodel-toolkit.git"
# Alternatively, the code used for simulations within this project can
# be retrieved by the command
# wget -"https://github.com/milasudril/snowflake-toolkit/archive/v1.0.tar.gz"
# and by extracting the content of that file

# Cd into the snowflake toolkit directory
cd "snowflakemodel-toolkit"

# Run Wand to compile the project
```

B Retrieving and compiling the toolkit

```
~/bin/wand "profile [ release ]"
```

During compilation of the toolkit, all targets will be written to `__wand_targets`, or `__wand_targets_dbg`. The targets include object files, that contains the compiled code for the toolkit, and executables. The executables are

snowflake_generate generate a snowflake using a static aggregate file

snowflake_prototype-test a test program for rendering of a single ice crystal prototype

snowflake_simulate implementation of model A

snowflake_simulate3 implementation of model B

All of these programs uses GNU style long options and accepts the option `--help`, which will print information about available parameters. As example the following command will print information about `snowflake_simulate3`:

```
__wand_targets/snowflake_simulate3 --help
```

C Predefined crystal files

Table C.1: Crystal prototypes bundled with the toolkit. The prototypes are viewed in orthographic projection, in the (x, y) -plane, and the (y, z) -plane. The parameters for `bullet.ice` and `hollow.ice` follow the same convention as Hong, G. (2007), table 1.

Name	Description of shape and parameters
<code>bullet.ice</code>	
<code>bullet-double.ice</code>	
<code>cuboid.ice</code>	
<code>hollow.ice</code>	
<code>spheroid.ice</code>	

D Command files used by the toolkit

The framework uses special command files to store input data, such as prototype crystal geometry, and aggregates. There are two kinds of command files. The first kind—aggregate files—is used to describe aggregates, and the second kind—crystal files—is used to describe prototype crystals.

D.1 General syntax

A command file contains a number of commands. An example of a command file can be found in listings D.1 and D.2. The syntax is based on single delimiter characters listed table D.1, each determining what action should be taken. Each command starts with its name, directly followed by an opening parenthesis `(` (U+0028). If a command is already being processed, the processing state is pushed onto a stack, so it can be finished later. Then the command arguments follow separated by comma `,` (U+002C). The argument list, and the command, end with a closing parenthesis `)` (U+0029). The preceding token is treated as a command argument, and the current command is executed. The result of the current command is then used as argument to the previous command.

A backslash `\` (U+005C) is used as escape character in the sense that the character that follows an escape character is interpreted literally. Any leading whitespace within a token is always skipped, unless it follows an escape character. The whitespaces that follow an escape character are always preserved. This also holds for

Table D.1: Delimiter characters and their function during the parsing process of command files. The corresponding Unicode codepoint is written in parenthesis in hexadecimal notation.

Character	Function
<code>#</code> (U+0023)	Indicates that a line is a comment
<code>(</code> (U+0028)	Starts a new command
<code>)</code> (U+0029)	Ends the current command and returns to the previous command
<code>,</code> (U+002C)	Delimits command arguments
<code>\</code> (U+005C)	Escape character

Table D.2: Commands allowed in an ice crystal prototype definition file

Command	Description
face	Defines a new face
group	Associates vertices with a deformation rule (see section 2.3.1)
matrix	Defines a new deformation rule (see section 2.3.1)
mirror_heading	Activates mirroring in the heading (<i>x</i>) direction
vertex	Defines vertex
volume	Starts a new sub-volume

whitespaces that follow a regular character in a token string. The characters that follow a number sign # (U+0023) are comments. The comment ends with the next line. Both a carriage-return and line-feed character are treated as line delimiters.

D.2 Ice crystal prototype definition files

Crystal definition files may contain the commands listed in table D.2. The `vertex` command adds a vertex to the current sub-volume. The location of the vertex are given as *x*, *y* and *z*.

A face is defined by the `face` command. The `face` command takes three zero-based vertex indices, each referring to one of the vertices within the current sub-volume, and a flag that tells the `SolidLoader` whether or not the face is visible. It is important that all vertices referred to by the `face` command are defined before the face. In order to get the face normal correct, the triangle defining the face has to be traversed counter-clockwise.

The `volume` command marks a new sub-volume. The following `vertex` and `face` up to the next `volume` command is added to the current sub-volume. As mentioned in section 2.2, the connections of the vertices inside each sub-volume has to form convex set.

The `matrix` command adds a new deformation rule (see section 2.3.1). The first argument is the name of the deformation rule. The 16 remaining arguments are the individual matrix elements in row-wise order. If an argument is not a number, it is treated as a parameter name, that will take a value during runtime. The deformations are applied to the ice crystal prototype in the same order as the corresponding `matrix` commands in the crystal definition file.

The vertices affected by a deformation rule are defined by the `group` command. The first argument associates the group with a deformation rule. Following the name of the deformation rule, the indices of all affected vertices are listed. The vertex indexing follows the same convention as for the `face` command. A vertex

index may be present in more than one group.

In addition to the commands mentioned above, there is the command `mirror_heading`, that takes all deformed geometry and adds a mirrored copy of the geometry to its tail.

Listing D.1: Example of an ice crystal prototype definition. This listing shows the definition for `bullet.ice`

```
#SnowflakeModel Ice crystal Prototype

#Scaling matrices

#front is the tip vertex

matrix(front,
       t,0,0,0,
       0,1,0,0,
       0,0,1,0,
       0,0,0,1)

#back is the flat end

matrix(back,
       L,0,0,0,
       0,1,0,0,
       0,0,1,0,
       0,0,0,1)

#global scaling matrix

matrix($global,
       1,0,0,0,
       0,a,0,0,
       0,0,a,0,
       0,0,0,1)

#Mesh subvolumes. Each of these needs to be convex

volume()
  vertex(-1,0,-5.960464e-08)
  vertex(1,0,5.960464e-08)
  vertex(-1,1,-5.960464e-08)
  vertex(0,1,5.960464e-08)
  vertex(-1,0.5,-0.8660255)
  vertex(1.192093e-07,0.5,-0.8660254)
  vertex(-1,-0.5000001,-0.8660254)
  vertex(1.192093e-07,-0.5000001,-0.8660253)
```

```
vertex(-1,-1,0)
vertex(0,-1,1.788139e-07)
vertex(-1,-0.4999999,0.8660254)
vertex(0,-0.4999999,0.8660255)
vertex(-1,0.4999999,0.8660254)
vertex(0,0.4999999,0.8660255)

face(0,2,4,outside)
face(1,5,3,outside)
face(3,5,4,outside)
face(0,4,6,outside)
face(1,7,5,outside)
face(5,7,6,outside)
face(0,6,8,outside)
face(1,9,7,outside)
face(7,9,8,outside)
face(0,8,10,outside)
face(1,11,9,outside)
face(9,11,10,outside)
face(0,10,12,outside)
face(1,13,11,outside)
face(11,13,12,outside)
face(0,12,2,outside)
face(1,3,13,outside)
face(13,3,2,outside)
face(2,3,4,outside)
face(4,5,6,outside)
face(6,7,8,outside)
face(8,9,10,outside)
face(10,11,12,outside)
face(12,13,2,outside)

group(back,0,2,4,6,8,10,12)
group(front,1)
```

D.2.1 Using Blender for creating ice crystal prototypes

Although it is possible to create ice crystal prototypes in a text editor, it is difficult to visualise the resulting geometry that way. As an alternative to creating ice crystal prototypes “by hand”, it is possible to draw them in **Blender**. This requires installing the **Blender** add-ons `io_scene_snowflakecrystalprototype` and `mesh_faces_tag`, both provided by the toolkit. After these add-ons has been installed *and* enabled, it is possible to use **Blender** as an ice crystal prototype editor.

Drawing sub-volumes

Each sub-volume required by the ice crystal prototype geometry is drawn as free-standing “mesh objects” (in **Blender** terminology). It is important to keep these mesh objects convex. Otherwise, the drawing procedure is identical to how 3D objects are drawn in **Blender**. Currently, the division of a solid into convex sub-volume has to be done manually, so it easiest to create the sub-volumes one by one. Vertex groups are used to bind vertices with a deformation.

Adding deformations

Deformations are added to the ice crystal prototype by adding a mandatory text resource, called **matrices**, to the **Blender** file. The text resource contains **matrix** commands that are copied when exporting the file to an ice crystal prototype. Even though the ice crystal prototype does not have any deformation, there must be a text resource called **matrices** before the **Blender** file is exported. Otherwise, the exporter will complain.

Triangulating faces

Before exporting the **Blender** file to an ice crystal prototype, all faces has to be split into triangles. This can be done automatically by using the “Triangulate” option from the mesh menu. It is a good idea to do triangulation before face tagging (see below), since face tags are not copied to the new faces created during triangulation.

Tagging faces

The **mesh_faces_tag** add-on, accessed from the “Misc” tab in the edit toolshelf, is used to tag faces, so the system knows whether or not a face will be visible. Visible faces are those that are not co-planar with another face. Such faces must have the tag **outside**. All other faces must have the tag **inside**.

D.3 Aggregate description files

Aggregate graphs can be loaded from an aggregate description file. An aggregate description file may contain the commands listed in table D.3. An example of an aggregate description file is shown in listing D.2.

D.3.1 Defining the graph

In order to create an aggregate graph, at least one crystal prototype file has to be loaded. The **crystal** command loads an ice crystal prototype file (see appendix D.2)

Listing D.2: Example of an aggregate description file. This listing shows the description file `bulletrosette-6.flake`.

```

param_declare(L,2)
param_declare(t,0.5)
param_declare(a,1)

crystal(bullet, .. / crystal-library/bullet.ice)

node(a, bullet)
node(b, bullet)
node(c, bullet)
node(d, bullet)
node(e, bullet)
node(f, bullet)

vector(u, 1, 0, 0)
nodes_connect(a, u, b, u, 0, 0, 0.5)
nodes_connect(a, u, c, u, 0, 0, 0.25)
nodes_connect(a, u, d, u, 0, 0, -0.25)
nodes_connect(a, u, e, u, 0, 0.25, 0)
nodes_connect(a, u, f, u, 0, -0.25, 0)

node_param_set(a,L,param_get(L))
node_param_set(b,L,param_get(L))
node_param_set(c,L,param_get(L))
node_param_set(d,L,param_get(L))
node_param_set(e,L,param_get(L))
node_param_set(f,L,param_get(L))

node_param_set(a,a,param_get(a))
node_param_set(b,a,param_get(a))
node_param_set(c,a,param_get(a))
node_param_set(d,a,param_get(a))
node_param_set(e,a,param_get(a))
node_param_set(f,a,param_get(a))

node_param_set(a,t,param_get(t))
node_param_set(b,t,param_get(t))
node_param_set(c,t,param_get(t))
node_param_set(d,t,param_get(t))
node_param_set(e,t,param_get(t))
node_param_set(f,t,param_get(t))

```

Table D.3: Commands allowed in an aggregate description file

Command	Description
crystal	Loads an ice crystal prototype
node	Associates a node name with an ice crystal
nodes_connect	Connects two nodes in an aggregate graph
node_param_set	Sets node parameters
vector	Defines an offset vector
param_declare	Defines a parameter that can be set from elsewhere
param_get	Retrieves the value of a previously declared parameter
add	Adds two numbers and returns their sum
curt	Takes the cubic root of its argument
div	Divides two numbers and returns their quotient
mult	Multiplies two numbers and returns their product
pi	Returns π
prod	Multiplies its arguments and returns their product
sub	Subtracts two numbers and substitutes returns their difference
print	Prints its arguments to the standard error stream

and associates the loaded ice crystal prototype with a name. The first argument is the symbolic name, and the second argument is the path of the file describing the ice crystal prototype. The symbolic name must be unique within the file. The path can be absolute, or relative to the directory of the current file.

The nodes in the aggregate graph are all associated with an ice crystal prototype. In order to associate a node with an ice crystal prototype, the command `node` is used. Its first argument is the name of the node, and the second argument is the symbolic name of a previously loaded ice crystal prototype. Each node within the file has to have a unique name, but they can share their ice crystal prototype.

Two nodes in the aggregate graph are connected by using the command `nodes_connect`. This command takes seven parameters: the first node and its offset vector, the second node and its offset vector, and finally the three rotation angles, divided by 2π . These parameters are all illustrated in fig. 2.3. Both the nodes and offset vectors has to be defined earlier in the file.

The deformation parameters (see section 2.3.1) of the ice crystal prototype associated with a node can be changed by the command `node_param_set`. The first argument of this command is the name of the affected node, and the two following are the parameter name and its new value, respectively.

Offset vectors (see section 2.3), are defined by the command `vector`. It takes four parameters, its symbolic name for use within the file, followed by the x , y , and z

coordinates.

D.3.2 Declaring and retrieving parameters

It is possible to pass parameters to the aggregate graph loading routine. Which parameters it takes is controlled by the use of the command `param_declare`. The command takes two parameters: its name, and its default value. The name of the parameter has to be unique within the current file.

Parameter values are retrieved through the command `param_get` command. The only parameter given to this command is the parameter name.

D.3.3 Arithmetical transformation of parameters

There is some basic support for arithmetical transformation of values. Through this mechanism, more complicated parametrisations can be used. The arithmetic functions area

- `add` that returns the sum of its arguments
- `curt` that returns the cubic root of its argument
- `div` that returns the quotient of its arguments. The first argument is the numerator.
- `mult` that returns the product of its arguments
- `pi` that returns π
- `prod` that returns the product of all of its arguments
- `sub` that returns the difference between its arguments

D.3.4 Other commands

In addition to the commands mentioned above, there is a `print` command, that prints its argument to the standard error stream. This command can be useful when debugging.

E API reference

This appendix gives a source-level overview of the toolkit, by listing files, classes and namespace-level functions. All identifiers that do not refer to a standard type are declared within the namespace `SnowflakeToolkit`.

E.1 List of source files

This list gives a short description of each source file in the toolkit.

- aggregate_edge.h** the definition of `AggregateEdge` (see appendix E.2.1)
- aggregate_graph.cpp** implementations of member functions of `AggregateGraph` (see appendix E.2.2)
- aggregate_graph.h** the definition of `AggregateGraph` (see appendix E.2.2)
- aggregate_graph_loader.cpp** the implementation of `AggregateGraphLoader` (see appendix E.2.3)
- aggregate_graph_loader.h** the definition of `AggregateGraphLoader` (see appendix E.2.3)
- aggregate_node.cpp** implementations of member functions of `AggregateNode` (see appendix E.2.4)
- aggregate_node.h** the definition of `AggregateNode` (see appendix E.2.4)
- bounding_box.h** the definition of `BoundingBox` (see appendix E.2.5)
- config_command.h** the definition of `ConfigCommand` (see appendix E.2.7)
- config_commandhandler.h** the definition of `ConfigCommandHandler` (see appendix E.2.6)
- config_parser.cpp** implementations of member functions of `ConfigParser` (see appendix E.2.8)
- config_parser.h** the definition of `ConfigParser` (see appendix E.2.8)
- element_randomizer.cpp** the implementation of member functions of `ElementRandomizer` (see appendix E.2.9)
- element_randomizer.h** the definition of `ElementRandomizer` (see appendix E.2.9)
- file_in.h** the definition of `FileIn` (see appendix E.2.10)

file_out.h the definition of **FileOut** (see appendix E.2.11)

ice_particle.cpp implementations of member functions of **IceParticle** (see appendix E.2.12)

ice_particle.h the definition of **IceParticle** (see appendix E.2.12)

ice_particle_vistor.h the definition of **IceParticleVisitor** (see appendix E.2.13)

matrix_storage.cpp implementations of member functions of **MatrixStorage** (see appendix E.2.14)

matrix_storage.h the defintion of **MatrixStorage** (see appendix E.2.14)

new.cpp replacement of operator new, and operator delete

profile.cpp implementations of member functions of **TicToc** (see appendix E.2.23)

profile.h simple facilities for profiling a task, and the definition of **TicToc** (see appendix E.2.23)

snowflake_generate.cpp aggregate graph assembly program

snowflake_prototype-test.cpp program for testing ice crystal prototypes

snowflake_simulate3.cpp implementation of model B

snowflake_simulate.cpp implementation of model A, but with simplified area calculation

solid_builder_bbc.cpp implementations of member functions of **SolidBuilderBBC** (see appendix E.2.16)

solid_builder_bbc.h definition of **SolidBuilderBBC** (see appendix E.2.16)

solid_builder.cpp implementations of member functions of **SolidBuilder** (see appendix E.2.15)

solid_builder.h definition of **SolidBuilder** (see appendix E.2.15)

solid.cpp implementations of member functions of **Solid** (see appendix E.2.17)

solid_deformation.h definition of **SolidDeformation** (see appendix E.2.18)

solid.h definition of **Solid** (see appendix E.2.17)

solid_loader.cpp implementations of member functions of **SolidLoader** (see appendix E.2.19)

solid_loader.h definition of **SolidLoader** (see appendix E.2.19)

solid_writer.cpp implementations of member functions of **SolidWriter** (see appendix E.2.20)

solid_writer.h definition of **SolidWriter** (see appendix E.2.20)

task.h definition of **Task** (see appendix E.2.21)

thread.cpp implementations of member functions of **Thread** (see appendix E.2.22)
thread.h definition of **Thread** (see appendix E.2.22)
twins.h definition of **Twins** (see appendix E.2.24)
vector.cpp implementation of geometry related functions
vector.h geometry related definitions
volume_convex.cpp implementations of member functions of **VolumeConvex** (see appendix E.2.25)
volume_convex.h definition of **VolumeConvex** (see appendix E.2.25)
voxelbuilder_adda.cpp implementations of member functions of **VoxelbuilderAdda** (see appendix E.2.28)
voxelbuilder_adda.h definition of **VoxelbuilderAdda** (see appendix E.2.28)
voxel_builder.h definition of **VoxelBuilder** (see appendix E.2.27)

E.2 Class reference

This section gives an overview of all classes defined within this toolkit. Each class is described in more detail in its own section. Some classes have **mutable** members. These members are used to store values that are expensive to compute. Upon request, their value is recomputed only if needed. In the current implementation, there is no mutex guarding access to these members. Therefore, the functions that retrieves such a member cannot be used from more than one thread at once.

struct AggregateEdge a connection between two nodes in an **AggregateGraph** graph.
See: appendix E.2.1

class AggregateGraph the class used for intermediate storage of ice particles.
See: appendix E.2.2

class AggregateGraphLoader:public ConfigCommandHandler the class used to load aggregate definition files
see appendix E.2.3

class AggregateNode a node in an **AggregateGraph** graph. *See:* appendix E.2.4

struct BoundingBox a description of a bounding box.
See: appendix E.2.5

class ConfigCommandHandler interface for processing events issued by a **ConfigParser**.
See: appendix E.2.6

struct ConfigCommand data passed from a **ConfigParser** to a **ConfigCommandHandler**.

See: appendix E.2.7

class ConfigParser class used for decoding configuration files described in appendix D.

See: appendix E.2.8

class ElementRandomizer class used to select a random element in a matrix.

See: appendix E.2.9

class Face a triangle found in a **VolumeConvex**. This class is defined inside **VolumeConvex**.

See: appendix E.2.25

class FileIn a thin wrapper class for reading files by using the C I/O API.

See: appendix E.2.10

class FileOut a thin wrapper class for writing files through the C I/O API.

See: appendix E.2.11

class IceParticle a building block for ice particles.

See: appendix E.2.12

class IceParticleVisitor interface for processing nodes in an **AggregateGraph** graph.

See: appendix E.2.13

class MatrixStorage a matrix without support for matrix algebra.

See: appendix E.2.14

class SolidBuilder:**public** **IceParticleVisitor** **IceParticleVisitor** that uses “regular” offset vectors
see appendix E.2.15

class SolidBuilderBBC:**public** **IceParticleVisitor** **IceParticleVisitor** that uses offset vectors normalised to the bounding box of the current **IceParticle**
see appendix E.2.16

class Solid a collection of sub-volumes used to store the geometry of an **IceParticle**
(see appendix E.2.12).

See: appendix E.2.17

class SolidDeformation a named 4×4 matrix with named elements.

See: appendix E.2.18

class SolidLoader:**public** **ConfigCommandHandler** class used to load a **Solid** from a ice crystal prototype file.
See: appendix E.2.19

class SolidWriter class used to store a **Solid** in Wavefront (FileFormat.Info 2015) file format.

See: appendix E.2.20

class Task interface for a running entity.

See: appendix E.2.21

class Thread class for initializing and synchronizing parallel tasks.

See: appendix E.2.22

class TicToc a timestamp manager.

See: appendix E.2.23

struct Twins:public std::pair<T,T> a pair whose members both have the type **T**.

See: appendix E.2.24

class VolumeConvex the most fundamental building block in ice particles.

See: appendix E.2.25

class VoxelBuilder interface for processing events issued when rasterising a **Solid**.

class VolumeConvex::Face Defines a face.

See: appendix E.2.26 *See:* appendix E.2.27

class VoxelbuilderAdda:public VoxelBuilder rasteriser writing data in ADDA format.

See: appendix E.2.28

E.2.1 struct AggregateEdge

An **AggregateEdge** is a connection between two nodes in an **AggregateGraph** (see appendix E.2.2) graph. The connection describes the bond connecting the associated **IceParticles**. The struct **AggregateEdge** has the following public members:

AggregateEdge (AggregateNode* node_parent,const Vector& offset_parent , AggregateNode* node_child,const Vector& offset_child , float angle_x,float angle_y, float angle_z) constructor initialising an **AggregateEdge**

Vector m_offset_parent the vector of the anchor point in the parent **IceParticle**

Vector m_offset_child the vector of the anchor point in the child **IceParticle**

AggregateNode* r_node_parent a pointer to the parent node

AggregateNode* r_node_child a pointer to the child node

float m_angle_x the roll angle

float m_angle_y the pitch angle

float m_angle_z the yaw angle

E.2.2 class AggregateGraph

An **AggregateGraph** is used for intermediate storage of ice particles. The storage model uses a graph structure similar to the one outlined in section 2.2.1. The public members of the class **AggregateGraph** are

AggregateGraph() constructor initialising an **AggregateGraph**. All it does, is setting the root point to the origin, and setting all angles to zero.

void nodesVisit(IceParticleVisitor&& builder) traverses all nodes in depth-first order.

**AggregateNode& nodeAppend(const Vector& u
, AggregateNode& node_v, const Vector& v
, float angle_x, float angle_y, float angle_z)** appends a new node to the graph, and connects it to `node_v`. The function returns a reference to the newly appended node.

AggregateNode& nodeAppend() appends a new node to the graph, without creating any connection. The function returns a reference to the newly appended node.

const Vector& positionGet() const noexcept returns the position of `*this` aggregate

Other members are

std::vector< std::unique_ptr<AggregateNode> > m_nodes all nodes in the graph

Vector m_position the position of the aggregate

float m_angle_root_x the roll angle of the aggregate

float m_angle_root_y the pitch angle of the aggregate

float m_angle_root_z the yaw angle of the aggregate

E.2.3 class AggregateGraphLoader:public ConfigCommandHandler

This kind of **ConfigCommandHandler** loads an **AggregateGraph** (see appendix E.2.2) from a configuration file. An **AggregateGraphLoader** accepts the commands listed in appendix D.3. The class **AggregateGraphLoader** has the following public members:

**AggregateGraphLoader (AggregateGraph& graph
, const std::map<std::string, std::string>& varlist
, std::map<std::string, Solid>& solids)** constructor initialising an `AggregateGraphLoader`

std::string invoke(const ConfigCommand& cmd

, const FileIn& source) function override from **ConfigCommandHandler** (see appendix E.2.6). The **ConfigCommands** accepted by This function are described in appendix D.3.

std::map<std::string, std::string>::const_iterator varsBegin() const This function returns an iterator to the beginning list of variables loaded from the aggregate description file

std::map<std::string, std::string>::const_iterator varsEnd() const This function returns an iterator to the end list of variables loaded from the aggregate description file

Other members are

AggregateGraph& r_graph a reference to the **AggregateGraph** being loaded

const std::map<std::string, std::string>& r_varlist a reference variables, indexed by name, loaded before initialising *this

std::map<std::string, Solid>& r_solids all **Solids** loaded, indexed by name

std::map<std::string, std::string> m_varlist all variables, indexed by name, loaded from the aggregate description file

std::map<std::string, AggregateNode*> m_nodes all nodes, indexed by name, loaded from the aggregate description file

std::map<std::string, Vector> m_vectors all offset vectors, indexed by name, loaded from the aggregate description file

E.2.4 class AggregateNode

An **AggregateNode** holds information about a node in an **AggregateGraph** (see appendix E.2.2). An **AggregateNode** has a “color” property that can be used to test whether or not the node has been visited. Two **AggregateNodes** can be connected by using the function

void bondCreate(AggregateNode& node_u, const Vector& u, AggregateNode& node_v, const Vector& v, float angle_x, float angle_y, float angle_z)

The public members of **AggregateNode** are

AggregateNode() constructor initialising the node

AggregateEdge* bondsBegin() const returns a pointer to the first bond of the node

AggregateEdge* bondsEnd() const returns a pointer to the bond past the last bond of the node

unsigned int colorGet() const returns the node color
void colorToggle() toggles the node color
IceParticle& iceParticleGet() returns a reference to the **IceParticle** (see appendix E.2.12) of the node
bool leafIs() const returns true if and only if this node is a leaf

Other members are

IceParticle m_ice_particle the **IceParticle** (see appendix E.2.12) of the node
std::vector<AggregateEdge> m_bonds All bonds starting at the node
static constexpr unsigned int COLOR_BIT=1 bit mask for node color
unsigned int m_flags node flags

E.2.5 struct BoundingBox

A **BoundingBox** describes a bounding box. The public members of **BouundingBox** are

Vector m_min Vector representing the “smallest” point
Vector m_max Vector representing the “largest” point

E.2.6 class ConfigCommandHandler

This is an interface used when processing events issued by a **ConfigParser** (see appendix E.2.8). The public members of the interface are

**virtual std::string invoke(const ConfigCommand& command
, const FileIn& source)=0** This method is invoked each time a command occurs in *source*. The returned string is used as argument to the outer command.

E.2.7 struct ConfigCommand

A **ConfigCommad** is used to pass command data from a **ConfigParser** (see appendix E.2.8) to a **ConfigCommandHandler** (see appendix E.2.6). The public members are

std::string m_name the command name
std::vector<std::string> m_arguments the command arguments

E.2.8 class ConfigParser

This class is used for decoding configuration files described in appendix D. The public members are

ConfigParser(FileIn& source) constructor associating the **ConfigParser** with a **FileIn** (see appendix E.2.10).

void commandsRead(ConfigCommandHandler& handler) reads the content of the associated **FileIn** (see appendix E.2.10)

Other members are

FileIn& r_source a reference to the **FileIn** (see appendix E.2.10) being read

E.2.9 class ElementRandomizer

This class is used to select a random element in a matrix. The public members are

ElementRandomizer(const MatrixStorage& M) constructor associating the **ElementRandomizer** with the matrix used for the probability distribution

Twins<size_t> elementChoose(std::mt19937& randgen) draws a random element from the associated matrix by using the std::mt19937 random source

Other members are

const MatrixStorage& r_M the probability matrix

E.2.10 class FileIn

This class is a thin wrapper class for reading files by using the C I/O API. The public members are

explicit FileIn(const char* source) constructor initialising the object by opening the file referred to by *source*

explicit FileIn(FILE* source) constructor initialising the object by setting the internal **FILE** object pointer to *source*. This is useful for reading standard input.

const std::string& filenameGet() const returns the associated filename, if any. If no filename is associated with the file, the returned string is empty.

int getc() directly returns the value returned by **getc**

Other members are

FILE* file_in a pointer to the underlying **FILE** object

std::string m_filename the name of the source file

In addition to the above members, the class has a deleted copy constructor and a deleted copy assignment operator.

E.2.11 class FileOut

This class is a thin wrapper class for writing files through the C I/O API. The public members are

explicit FileOut(const char* dest) constructor initialising the object by opening the file referred to by *dest*

explicit FileOut(FILE* dest) constructor initialising the object by setting the internal **FILE** object pointer to *source*. This is useful for writing standard output or standard error.

const std::string& filenameGet() const returns the associated filename, if any.
If no filename is associated with the file, the returned string is empty.

void getc(char ch) directly calls `putc`

void printf(const char* format,...) printf-style function

Other members are

FILE* file_out a pointer to the underlying **FILE** object

std::string m_filename the name of the destination file

In addition to the above members, the class has a deleted copy constructor and a deleted copy assignment operator.

E.2.12 class IceParticle

An **IceParticle** is a building block for ice particles. The class, whose purpose is described in section 2.2, has the following public members

IceParticle() constructor initialising the object

void solidSet(const Solid& solid) associates a **Solid** (see appendix E.2.17) with the **IceParticle**. If the particle is dead, this resurrects the particle.

const Solid& solidGet() const This function returns the **Solid** (see appendix E.2.17) of the **IceParticle**. The **Solid** returned is the transformed version of the **Solid** set by `solidSet`.

Solid& solidGet() Non-const version of the function above

void parameterSet(const std::string& name, float value) This function sets a deformation parameter

void solidScale(float c) This function rescales the deformed version of the associated **Solid** (see appendix E.2.17) by the factor *c*

const Vector& velocityGet() const This function returns the current velocity of the **IceParticle**

void velocitySet(const Vector& v) const This function sets the velocity of the **IceParticle**

float densityGet() This function returns the current density of the **IceParticle**

void densitySet() This function sets the density of the **IceParticle**

void kill() This function marks the **IceParticle** as dead

bool dead() This function returns non-zero if and only if the **IceParticle** is dead

Also, the class has the following members:

const Solid* r_solid a pointer to the associated solid

mutable std::vector<SolidDeformation> m_deformations all **SolidDeformations** (see appendix E.2.18) that can be applied to **r_solid**

mutable Solid m_solid a **Solid** (see appendix E.2.17) that has been deformed using the deformations from **m_deformations**

mutable uint32_t m_flags_dirty bit field indicating which data members that needs to be recomputed

static constexpr uint32_t DEFORMATIONS_DIRTY=0x1 this bit mask is used to indicate that the associated deformations needs to be updated

static constexpr uint32_t VOLUME_DIRTY=0x2 this bit mask is used to indicate that **m_solid** needs to be regenerated

Vector m_velocity the particle velocity

float m_density the particle density

bool m_dead the dead status of the particle

void solidGenerate() const This function updates **m_solid** from **r_solid** and the deformations given by **m_deformations**

E.2.13 class IceParticleVisitor

This interface is used to define entry points needed for processing nodes in an **AggregateGraph** (see appendix E.2.2) graph. It has the following public members:

virtual void branchBegin(AggregateEdge& edge, AggregateGraph& graph)=0
this method is invoked each time the graph traversal algorithm begins a new branch

virtual void branchEnd(AggregateEdge& edge, AggregateGraph& graph)=0 this method is invoked each time the graph traversal algorithm completes a branch

virtual void iceParticleProcess(AggregateEdge& edge, AggregateGraph& graph)=0 this method is invoked each time the graph traversal algorithm needs to process a node

E.2.14 class MatrixStorage

This class is used to store a matrix, interpreted in as if its elements are stored row-wise. Its public members are

typedef double ElementType The type of individual elements. When **ElementType** is used within this section, it refers to this type

static size_t N_validate(size_t N, size_t M) This function test whether or not the specified matrix size is valid. If the number is valid, it returns $M*N$, otherwise it throws an exception.

MatrixStorage(size_t N_rows, size_t N_cols) constructor initialising the matrix

const ElementType& operator()(size_t row, size_t col) const Element access

ElementType& operator()(size_t row, size_t col) the same as above but non-const version

void symmetricAssing(size_t row, size_t col, const ElementType& value) This function assings two matrix elements to value, keeping the matrix symmetric

const ElementType* rowGet(size_t row) const This function returns a pointer to the specified row

ElementType* rowGet(size_t row) const the same as above, but non-const version

size_t sizeGet() const This function returns the number of elements in the matrix

size_t nColsGet() const This function returns the number of columns in the matrix

size_t nRowsGet() const This function returns the number of rows in the matrix

const ElementType* rowsEnd() const This function returns a pointer to the row following the last row

ElementType* rowsEnd() the same as above, but non-const version

Twins<size_t> locationGet(size_t index) This function converts an element index to a row/column pair

ElementType sumGet() const This function returns the sum of all matrix elements. If the sum needs to be recomputed, it is done single-threaded.

ElementType sumGetMt() const This function returns the sum of all matrix elements. If the sum needs to be recomputed, it is done with multiple threads.

Other members are

typedef ElementType vec4_t __attribute__((vector_size(4*sizeof(double))))
vectorised double

```
size_t m_N_cols the number of columns
std::vector<ElementType> m_data the matrix data
mutable ElementType m_sum the sum of all matrix elements
class SumTask class describing an element summation task
mutable std::vector<ElementType> m_sums_row all partial sums computed
from worker Threads (see appendix E.2.22)
mutable std::vector<SumTask> m_sums all summation tasks
mutable std::vector<m_sum_workers> m_sum_workers summation worker
Threads
mutable uint32_t m_flags_dirty bit field indicating which values needs to be
recomputed
static constexpr uint32_t SUM_DIRTY=0x1 bit mask indicating that the ma-
trix sum needs to be updated
void sumCompute() const single-threaded sum computation
void sumComputeMt() const multi-threaded sum computation
```

E.2.15 class SolidBuilder:public IceParticleVisitor

A **SolidBuilder** is a **IceParticleVisitor** (see appendix E.2.13) that uses “regular” offset vectors. Regular in this context means that they are expressed in standard basis, as opposed to the basis defined by the bounding box of an **IceParticle**. The public members of **SolidBuilder** are

```
SolidBuilder(Solid& mesh_out) constructor initialising the SolidBuilder
void branchBegin(AggregateEdge& edge,AggregateGraph& graph) function over-
ride from IceParticleVisitor (see appendix E.2.13). This function pushes
the current bond onto the internal stack
void branchEnd(AggregateEdge& edge,AggregateGraph& graph) function over-
ride from IceParticleVisitor (see appendix E.2.13). This function restores
the current node from the internal stack.
void iceParticleProcess(AggregateEdge& edge,AggregateGraph& graph) func-
tion override from IceParticleVisitor (see appendix E.2.13). This function
follows the algorithm outlined in section 2.3 to assemble a Solid.
```

Other members are

```
Solid& r_mesh_out a reference to the Solid (see appendix E.2.17) being assem-
bled
```

struct Bond definition of a “bond”
Bond m_bond the current “bond”
std::stack<Bond> m_bonds all “bonds”

E.2.16 class SolidBuilderBBC:**public** IceParticleVisitor

This class is identical to **SolidBuilder** (see appendix E.2.15), but instead of standard basis, it uses “bounding box coordinates” for the offset vectors. That is $\mathbf{u} = (1, 1, 1)$ refers to a corner in the current **IceParticle**.

E.2.17 class Solid

This class is used to store the geometry data of an **IceParticle** (see appendix E.2.12). A **Solid** is composed of **VolumeConvexs** (see appendix E.2.25), and **SolidDeformations** (see appendix E.2.18). The class has the following public members:

static constexpr uint32_t MIRROR_HEADING=0x1 bit mask indicating that the solid should be mirrored in the plane with normal parallel to the heading direction

Solid() constructor initialising the object

VolumeConvex& subvolumeAdd(const VolumeConvex& volume) This function adds another **VolumeConvex** (see appendix E.2.25) to the object, and returns a reference to the newly appended **VolumeConvex**

VolumeConvex& subvolumeAdd(VolumeConvex&& volume) This function does the same as the one above, but is implemented with move semantics

const VolumeConvex* subvolumesBegin() const This function returns a pointer to the first sub-volume in the **Solid**

const VolumeConvex* subvolumesEnd() const This function returns a pointer to the sub-volume after the last

VolumeConvex* subvolumesBegin() This function returns a pointer to the first sub-volume in the **Solid**, non-const version

VolumeConvex* subvolumesEnd() This function returns a pointer to the sub-volume after the last, non-const version

size_t subvolumesCount() const This function returns the number of sub-volumes within the **Solid**

const VolumeConvex& subvolumeGet(size_t index) const This function returns a reference to the sub-volume at position **index**

VolumeConvex& subvolumeGet(size_t index) the same as above, but non-const version

void merge(const Matrix& T,const Solid& volume,bool mirrored) This function merges **volume** into this **Solid**, while applying the transformation matrix **T**. If this matrix has a negative determinant, the **mirrored** argument has to be non-zero

void merge(const Solid& volume) the same as above, but without any transformation

const BoundingBox& boundingBoxGet() const This function returns the bounding box of the **Solid**

const Point& midpointGet() const This function returns the centroid of the **Solid**. The returned value is only correct if none of the sub-volumes overlap each other.

float volumeGet() const This function returns the volume of the **Solid**. The returned value is only correct if none of the sub-volumes overlap each other.

float rMaxGet() const This function returns the maximal distance from the centroid of the **Solid** to its edge

void geometrySample(VoxelBuilder& builder) const This function samples the geometry following the algorithm described in section 2.4. Visited voxels are passed to the **VoxelBuilder** (see appendix E.2.27) referred to by **builder**.

void transform(const Matrix& T,bool mirrored) const This function transforms the **Solid** with the **Matrix** **T**. The parameter **mirrored** must be non-zero if the transformation causes the face normals to be reversed.

const VolumeConvex* inside(const Point& v) const This function tests whether or not the **Point& v** lies inside the **Solid**. If so, the function returns a pointer to the **VolumeConvex** that contains the point. Otherwise, the function returns **nullptr**.

const VolumeConvex* cross(const VolumeConvex::Face& face) const This function tests whether or not **face** crosses a face in the **Solid**. If so, the function returns a pointer to the **VolumeConvex** that contains the crossing **Face**. Otherwise, the function returns **nullptr**.

void centerCentroidAt(const Point& pos_new) This function moves the **Solid** in a way such that its centroid becomes located at **pos_new**.

void centerBoundingBoxAt(const Point& pos_new) This function moves the **Solid** in a way such that its bounding box becomes centred at **pos_new**.

void normalsFlip() const This function reverses the direction of all normal vectors

void deformationTemplateAdd(SolidDeformation&& deformation) This function adds a **SolidDeformation** (see appendix E.2.18) to the **Solid**

const std::vector<SolidDeformation>& deformationTemplatesGet() const This function returns a reference to all **SolidDeformations** (see appendix E.2.18) of this **Solid**

void mirrorActivate(uint32_t mirror_flags) This function activates mirroring for the directions specified by **mirror_flags**

bool mirrorFlagTest(uint32_t const This function tests whether or not any mirror is active

void mirrorDeactivate(uint32_t mirror_flags) the opposite of **mirrorActivate**

size_t facesCount() const This function returns the number of faces of the **Solid**

void clear() This function makes the **Solid** empty

Other members are

static constexpr uint32_t BOUNDINGBOX_DIRTY=0x1 this bit mask is used to test whether or not the bounding box of the **Solid** needs to be recomputed

static constexpr uint32_t MIDPOINT_DIRTY=0x2 this bit mask is used to test whether or not the centroid of the **Solid** needs to be recomputed

static constexpr uint32_t FACES_COUNT_DIRTY=0x4 this bit mask is used to test whether or not the number of faces needs to be recomputed

static constexpr uint32_t RMAX_DIRTY=0x8 this bit mask is used to test whether or not the maximal distance from the centroid the surface needs to be recomputed

static constexpr uint32_t VOLUME_DIRTY=0x10 this bit mask is used to whether or not the volume of the **Solid** needs to be recomputed

std::vector<VolumeConvex> m_subvolumes this member contains all sub-volumes. The sub-volumes are stored as **VolumeConvexs** (see appendix E.2.25).

std::vector<SolidDeformation> m_deformation_templates this member contains all **SolidDeformations** (see appendix E.2.18)

mutable BoundingBox m_bounding_box this is the bounding box of the **Solid**

mutable Point m_mid this is the centroid of the **Solid**

mutable size_t m_n_faces this is the number of faces of the **Solid**

mutable float m_r_max this is the maximal distance from the centroid of the **Solid** to its edge

mutable float m_volume this is the volume of the **Solid**

```
uint32_t m_mirror_flags bit field mirror flags  
void midpointCompute() const this is a helper routine  
void boundingBoxCompute() const this is a helper routine  
void facesCountCompute() const this is a helper routine  
void rMaxCompute() const this is a helper routine  
void volumeCompute() const this is a helper routine
```

E.2.18 class SolidDeformation

A **SolidDeformation** is a named 4×4 matrix with named elements. Elements with the same name can be accessed through the member function **parameterFind**. This function returns all column-wise element indices in the matrix, that have a name identical to the name given. The corresponding elements can then be updated through the **operator[]**. The class has the following public members:

```
SolidDeformation(const std::string& name) constructor initialising the deformation. Initially, the deformation is the identity matrix, and no matrix elements have an associated name.  
float operator[](size_t index) const returns the value at element at the given index as if the matrix were stored column-wise  
float& operator[](size_t index) returns a reference to the element at the given index as if the matrix were stored column-wise  
void& parameterDefine(const std::string& name,size_t index ,float value_default) This function assigns name to the element at index as if the matrix were stored column-wise. The parameter will have the default value given by value_default  
const std::vector<size_t> parameterFind(const std::string& name) const This function returns a pointer to a std::vector<size_t> containing all matrix element indices that matches name. If no index matches, nullptr is returned  
const std::string& nameGet() const This function returns the name of the SolidDeformation  
const Matrix& matrixGet() const This function returns the transformation matrix  
const std::string* parametersBegin() const This function returns a pointer to the name of the first parameter  
const std::string* parametersEnd() const This function returns a pointer to the name of the parameter past the last
```

Other members are

Matrix m_matrix the 4×4 transformation matrix. Elements are stored columnwise

std::string m_name the name of the deformation

std::map< std::string, std::vector<size_t> > parameter_map a `std::map` from parameter name to element index

std::vector<std::string> param_names a `std::vector` of all parameter names

E.2.19 class SolidLoader:public ConfigCommandHandler

A **SolidLoader** is a **ConfigCommandHandler** (see appendix E.2.6) that is used to load a **Solid** (see appendix E.2.17) from a ice crystal prototype file. The commands understood by a **SolidLoader** are described in appendix D.2. The public members of **SolidLoader** are

SolidLoader(Solid& solid) constructor that associates the **SolidLoader** with the output **Solid**

std::string invoke(const ConfigCommand& cmd,const FileIn& source) function override from **ConfigCommandHandler**. This function processes the **ConfigCommand** cmd according to the rules in appendix D.2.

Other members are

Solid& r_solid a reference to the target **Solid**

VolumeConvex* r_vc_current a pointer to the current sub-volume

E.2.20 class SolidWriter

A **SolidWriter** can be used to store a **Solid** in Wavefront (FileFormat.Info 2015) file format. The class has the following public members:

SolidWriter(FileOut& dest) constructor connecting the **SolidWriter** to the **FileOut** (see appendix E.2.11) object given by **dest**

void write(const Solid& solid) This function writes **solid** to the destination file

Other members are

FileOut& r_dest a reference to the destination file

E.2.21 class Task

A **Task** is a running entity. An object of a class implementing this interface can be used to define the execution of a **Thread** (see appendix E.2.22). If the **Task** maintains any internal state, it should probably not be passed to more than one **Thread**. The interface has the following public members:

virtual void run() noexcept=0 this method is invoked from inside a thread after its execution has started.

E.2.22 class Thread

This is a thin wrapper class around the POSIX thread type. The class has the following public members:

Thread& operator=(Thread&& obj) noexcept this is the move-assignment operator

Thread(Thread&& obj) noexcept this is the move constructor

Thread(Thread& task, uint32_t thread_count) this constructor initialises a new **Thread** and starts its execution in **Task::run** (called on **task**). The thread is only allowed to run on the CPU identified by **thread_count**.

static uint32_t threadsMax() This function returns the maximum number of simultaneous threads. The returned value is the same as **sysconf(_SC_NPROCESSORS_ONLN)**.

~Thread() this is the class destructor. It will freeze the calling thread until the managed thread has completed its operation.

Other members are

struct Impl struct describing the internal representation

Impl* pimpl this is a pointer to the internal representation

E.2.23 class TicToc

A **TicToc** can be used to profile a block of code. By creating a **TicToc** object within two curly braces, the time it takes execute the code between the braces can be measured. TicToc object shares a common file handle connected to the file **profile_data.txt** within the working director, so only one **TicToc** object can be used simultaneously. To profile code running in parallel, bind the **TicToc** object to the same code block as all parallel threads. The class has the following public members:

TicToc(const char* file,int line) constructor that saves *the pointer* file and line.
Also, it stores a timestamp given by the system.

~TicToc() the destructor prints the time difference between destruction and construction to the profile file.

Other members are

static FileOut s_stats the log file for all timestamps
const char* r_file the source file of the codeblock being profiled by the **TicToc**
size_t* m_line the line where the **TicToc** was constructed
double m_start the time when the control entered the code block being profiled

E.2.24 struct Twins:public std::pair<T,T>

This class is identical to `std::pair` (see cppreference.com (2015)) with the two types identical, hence the name **Twins**.

E.2.25 class VolumeConvex

The class **VolumeConvex** is the fundamental representation of ice particle geometry, and as the name suggest, the geometry described by a **VolumeConvex** needs to be convex. A **VolumeConvex** consists of a polygon mesh, represented by vertices and faces. A face can be marked and unmarked as visible. Visible faces are referred to from an internal array by indices that refers to the zero-based face number within the **VolumeConvex**. A **VolumeConvex** also keeps an associative map between vertex group names, and vertex indices. Like in the case of the visible faces, vertex indices refers to the vertex number within the **VolumeConvex**.

The class **VolumeConvex** has the following public members:

typedef uint16_t VertexIndex A type describing a vertex index. Since this type only reserves 16 bits, a **VolumeConvex** can only contain 65536 vertices, but that should not be a limitation, since a **Solid** (see appendix E.2.17) can hold **VolumeConvex**s only limited by the amount of available memory

typedef uint16_t FaceIndex A type describing a face index. Since this type only reserves 16 bits, a **VolumeConvex** can only contain 65536 faces, but that should not be a limitation, since a **Solid** (see appendix E.2.17) can hold **VolumeConvex**s only limited by the amount of available memory

static constexpr uint16_t VERTEX_COUNT=3 The number of vertices in a face

Point Vertex The type of a vertex

class Face Defines a face. **VolumeConvex::Face** (see appendix E.2.26)

VolumeConvex() Default constructor initialising an empty **VolumeConvex**

VolumeConvex(const VolumeConvex& vc) Class copy constructor

size_t vertexAdd(const Matrix& T, const Point& p) Adds a transformed vertex to the **VolumeConvex** and returns the new number of vertices.

size_t vertexAdd(const Point& p) This function adds a vertex to the **VolumeConvex** and returns the new number of vertices.

const Vertex* verticesBegin() const This function returns a pointer to the first vertex within the **VolumeConvex**

const Vertex* verticesEnd() const This function returns a pointer to the vertex past the last within the **VolumeConvex**

const Vertex* verticesBegin() This function returns a pointer to the first vertex within the **VolumeConvex**

const Vertex* verticesEnd() This function returns a pointer to the vertex past the last within the **VolumeConvex**

size_t verticesCount() const This function returns the number of vertices within the **VolumeConvex**

const Vertex& vertexGet(size_t index) const This function returns a reference to the vertex at position **index**

void faceAdd(const Face& face) This function adds a new face to the **VolumeConvex**

const Face* facesBegin() const This function returns a pointer to the first face within the **VolumeConvex**

const Face* facesEnd() const This function returns a pointer to a face past the last within the **VolumeConvex**

Face* facesBegin() This function returns a pointer to the first face within the **VolumeConvex**

Face* facesEnd() This function returns a pointer to the face past the last within the **VolumeConvex**

const Face& faceGet(size_t index) const This function returns a reference to the **VolumeConvex::Face** (see appendix E.2.26) at position **index**

void vertexGroupSet(std::string& name, VertexIndex index) This function adds the vertex with index **index** to the vertex group **name**

const BoundingBox& boundingBoxGet() const This function returns the axis aligned bounding box of the **VolumeConvex**

const Point& midpointGet() This function returns the centroid of the **VolumeConvex**.
For the formula used, see section 2.5.4

float volumeGet() const This function returns the volume of the **VolumeConvex**.
For the formula used, see section 2.5.4

float areaVisibleGet() const This function returns the total visible area of the **VolumeConvex**

void transform(const Matrix& T) This function applies the transformation matrix **T** to the entire **VolumeConvex**. If **T** has a negative determinant, the function **normalsFlip** needs to be called afterwards.

void transformGroup(std::string& name, const Matrix& T) This function applies the transformation matrix **T** to the vertex group **name VolumeConvex**

bool inside(const Point& point) const This function test whether or not point is inside the **VolumeConvex**. The algorithm used to perform the test is explained in section 2.5.3.

const Face* cross(const Face& face) const This function tests whether or not the **VolumeConvex::Face** (see appendix E.2.26) referred to by **face** crosses any face within this **VolumeConvex**. If there is such a face, a pointer to the crossing **Face** is returned. Otherwise, the function returns **nullptr**. The algorithm used to implement this function is described in section 2.5.5.

void geometrySample(VoxelBuilder& builder) const This function samples the geometry of the **VolumeConvex** following the algorithm described in section 2.4. Visited voxels are passed to the **VoxelBuilder** (see appendix E.2.27) referred to by **builder**.

void normalsFlip() This function flips the direction of all face normals

void facesNormalCompute() const This function updates the face normals

void facesMidpointCompute() const This function updates the midpoint of all faces

void boundingBoxCompute() const This function updates the bounding box

void midpointCompute() const This function updates the centroid

void volumeCompute() const This function updates the volume

void areaVisibleCompute() const This function updates the visible surface area

FaceIndex facesOutCount() const This function returns the number of visible faces in the **VolumeConvex**

const FaceIndex* facesOutBegin() const This function returns a pointer to the first face index in the array of visible faces.

const FaceIndex* facesOutEnd() const This function returns a pointer to the face index past the last in the array of visible faces.

void faceOutAdd(FaceIndex i) This function adds face index **i** to the array of visible faces

const Face& faceOutGet(size_t i) const This function returns a reference to the face that corresponds to the visible face *i*

bool normalsDirty() const This function returns true whenever any of the face normals needs to be updated

It also has the following private members:

std::vector<Vertex> m_vertices contains all vertices of the **VolumeConvex**

std::vector<Face> m_faces contains all faces of the **VolumeConvex**

std::vector<FaceIndex> m_faces_out contains the indices of all visible faces of the **VolumeConvex**

std::map<std::string, std::vector<VertexIndex> > m_faces_out associates a vertex group with vertices

mutable BoundingBox m_bounding_box is the current **BoundingBox** (see appendix E.2.5) of the **VolumeConvex**

mutable Point m_mid is the current centroid of the **VolumeConvex**

mutable float m_volume is the current volume of the **VolumeConvex**

mutable float m_area_visible is the current visible area of the **VolumeConvex**

mutable uint32_t m_flags_dirty is a bit field indicating what needs to be updated

static constexpr uint32_t BOUNDINGBOX_DIRTY=0x1 This bit mask indicates that the bounding box needs to be recomputed

static constexpr uint32_t MIDPOINT_DIRTY=0x2 This bit mask indicates that the centroid needs to be recomputed

static constexpr uint32_t FACES_NORMAL_DIRTY=0x4 This bit mask indicates that the face normals need to be recomputed

static constexpr uint32_t FACES_MIDPOINT_DIRTY=0x8 This bit mask indicates that the face midpoints need to be recomputed

static constexpr uint32_t VOLUME_DIRTY=0x10 This bit mask indicates that the volume needs to be recomputed

static constexpr uint32_t AREA_VISIBLE_DIRTY=0x20 This bit mask indicates that the visible area needs to be recomputed

E.2.26 class VolumeConvex::Face

A **Face** consists of three vertices in counter-clockwise order, and a reference to the **VolumeConvex** (see appendix E.2.25) owning the **Face**. The class has the following public members:

Face(VertexIndex v0,VertexIndex v1,VertexIndex v2,const VolumeConvex& parent) Constructor initialising the Face.

const Vertex vertexGet(int index) const This function returns a reference to the vertex with the given index, which has to be less than VERTEX_COUNT, which equals 3.

mutable Point m_mid is the median point of the face

mutable Vector m_normal is the normalised face normal vector \hat{n}

mutable Vector m_normal_raw is the non-normalised face normal vector n

void directionChange() This function changes the orientation of the face. Notice that after calling this function, the face normal vector needs to be updated.

VertexIndex vertexIndexGet(int index) const returns the index (in the associated VolumeConvex (see appendix E.2.25)) of the given vertex index, which has to be less than VERTEX_COUNT, which equals 3.

void parentSet(const VolumeConvex& vc) This functions associates a new VolumeConvex (see appendix E.2.25) with the face, potentially invalidating the vertex references. Its existence is mainly for the copy constructor of VolumeConvex.

It also has the following private members:

VertexIndex m_verts[VERTEX_COUNT] The indices of the vertices defining the face

VolumeConvex* r_vc A pointer to the current VolumeConvex (see appendix E.2.25)

E.2.27 class VoxelBuilder

This interface defines events issued when rasterising a Solid. It public members are

virtual bool fill(const PointInt& position)=0 this method is invoked for each voxel visited. If no more voxel within the current sub-volume should be processed, the implementation shall return zero.

virtual void volumeStart(const VolumeConvex& volume_new)=0 this method is invoked every time a new sub-volume is visited

virtual PointInt quantize(const Point& position)=0 this method is invoked every time “real” coordinates should be converted to discrete coordinates

virtual Point dequantize(const PointInt& position)=0 the opposite of quantize

E.2.28 class VoxelBuilderAdda:**public** VoxelBuilder

This kind of **VoxelBuilder** (see appendix E.2.27) writes voxel data formatted as an a-dda¹ geometry file. The class has the following public members:

**VoxelBuilderAdda(FileOut& dest,int n_x,int n_y,int n_z
,const BoundingBox& bounding_box)** Constructor initialising the **VoxelBuilderAdda** with a reference to the output file, the number of raster points, and the **BoundingBox** (see appendix E.2.5) of the volume to fill. If any of **n_x**, **n_y**, and **n_z** is less than or equal to zero, its value is computed to preserve the aspect ratio of the boudning box. The number of raster points should be chosen with care too few raster points will result in an incomplete representation of the sampled geometry, but to many raster points will result in a larger computation time.

virtual bool fill(const PointInt& position)=0 function override from **VoxelBuilder** (see appendix E.2.27)

virtual void volumeStart(const VolumeConvex& volume_new)=0 function override from **VoxelBuilder** (see appendix E.2.27). This function resets **m_data_stop** and sets **r_volume_current** so it points to **volume_new**.

virtual PointInt quantize(const Point& position)=0 function override from **VoxelBuilder** (see appendix E.2.27). This function is implemented as eq. (2.1).

virtual Point dequantize(const PointInt& position)=0 function override from **VoxelBuilder** (see appendix E.2.27). This function is implemented as eq. (2.2).

~VoxelBuilderAdda() The class destructor

Other members are

FileOut& r_dest a reference to the **FileOut** (see appendix E.2.11) being written
uint8_t* m_data_filled a pointer to an array holding information about all voxels that has been filled

uint8_t* m_data_stop a pointer to an array holding information which voxels that has been visited within the current sub-volume

BoundingBox m_bounding_box the bounding box to fill

const VolumeConvex* r_volume_current a pointer to the current sub-volume.

int m_n_x the number of raster points in the *x* direction

int m_n_y the number of raster points in the *y* direction

int m_n_z the number of raster points in the *z* direction

¹See <http://code.google.com/p/a-dda>. At the time of writing, the project is being moved to <https://github.com/adda-team/adda>

E.3 Function reference

This is a list of all free functions within the toolkit

void bondCreate(AggregateNode& node_u,const Vector& u, AggregateNode& node_v,const Vector& v,float angle_x,float angle_y,float& angle_z)

This function creates a bond between the **AggregateNodes** (see appendix E.2.4) **node_u** and **node_v** with the corresponding offset vectors. For an illustration of the parameters, see fig. 2.3.

inline std::pair<float,uint32_t> extentMax(const Vector& v) This function looks for the largest component in a **Vector** and returns its value together with its zero-based index

std::pair<Matrix, bool> vectorsAlign(const Vector& dir

, const Vector& dir_target) This function computes the matrix needed for transforming the **unit vector** **dir** so it points in the direction of the **unit vector** **dir_target**. In the case the determinant of the **Matrix** returned **std::pair** is negative, its second member is non-zero. The formula used to generate the transformation matrix is found in section 2.5.2.

F Result tables

Table F.1: Computed values of the fractal dimension β of particles generated by model B for different event rates. Parameters not listed in the table are those found in table 4.2. The values were computed following the procedure in section 4.5.1.

R_g	R_m	R_d	β_{cloud}	$\Pr(\beta_{\text{cloud}})$	β_{dropped}	$\Pr(\beta_{\text{dropped}})$
1.5×10^7	5×10^2	1×10^4	2.03	13.9	1.98	12.7
1.5×10^7	5×10^2	2×10^4	2.05	15.7	2	10.1
1.5×10^7	5×10^2	5×10^4	1.81	38	1.73	11
1.5×10^7	1×10^3	1×10^4	1.94	19.9	1.93	66.4
1.5×10^7	1×10^3	2×10^4	2.02	6.37	1.97	25.3
1.5×10^7	1×10^3	5×10^4	1.75	11.3	1.68	8.36
1.5×10^7	2×10^3	1×10^4	2.15	14.8	2	10.3
1.5×10^7	2×10^3	2×10^4	1.78	18	1.82	42
1.5×10^7	2×10^3	5×10^4	1.46	52.8	1.54	42
2.2×10^7	5×10^2	1×10^4	2.01	18.3	1.96	42.7
2.2×10^7	5×10^2	2×10^4	1.93	13.8	1.95	7.8
2.2×10^7	5×10^2	5×10^4	1.81	22.1	1.89	41.6
2.2×10^7	1×10^3	1×10^4	2.04	6.73	2.02	24.8
2.2×10^7	1×10^3	2×10^4	2.15	7.59	2.02	4.54
2.2×10^7	1×10^3	5×10^4	1.77	30.1	1.81	11.5
2.2×10^7	2×10^3	1×10^4	2.08	18.5	2.03	30.4
2.2×10^7	2×10^3	2×10^4	1.95	14.6	1.96	5.45
2.2×10^7	2×10^3	5×10^4	1.72	13.3	1.69	23.8
3.3×10^7	5×10^2	1×10^4	1.9	16	2.04	47.8
3.3×10^7	5×10^2	2×10^4	2.03	14.1	1.95	10.2
3.3×10^7	5×10^2	5×10^4	1.93	20.2	1.86	10.7
3.3×10^7	1×10^3	1×10^4	1.88	9.93	1.95	12.5
3.3×10^7	1×10^3	2×10^4	1.97	19.3	1.97	9.72
3.3×10^7	1×10^3	5×10^4	1.84	10.4	1.91	15.8
3.3×10^7	2×10^3	1×10^4	1.92	4.25	1.97	122
3.3×10^7	2×10^3	2×10^4	1.98	7.93	1.97	52.5
3.3×10^7	2×10^3	5×10^4	1.91	17.8	1.81	8.57

Table F.2: Computed values of the normalised volume growth coefficient α of particles generated by model B for different event rates. Parameters not listed in the table are those found in table 4.2. The values were computed following the procedure in section 4.5.1.

R_g	R_m	R_d	α_{cloud}	$\Pr(\alpha_{\text{cloud}})$	α_{dropped}	$\Pr(\alpha_{\text{dropped}})$
1.5×10^7	5×10^2	1×10^4	0.964	3.08	1.03	15.3
1.5×10^7	5×10^2	2×10^4	0.825	13.8	0.909	2.84
1.5×10^7	5×10^2	5×10^4	1.02	37.7	1.13	14.5
1.5×10^7	1×10^3	1×10^4	1.06	22.9	1.09	104
1.5×10^7	1×10^3	2×10^4	0.819	3.71	0.918	11.5
1.5×10^7	1×10^3	5×10^4	1.04	5.88	1.13	11.6
1.5×10^7	2×10^3	1×10^4	0.687	13.8	0.878	6.96
1.5×10^7	2×10^3	2×10^4	1.03	12.6	1.03	24.3
1.5×10^7	2×10^3	5×10^4	1.24	65.4	1.2	35.3
2.2×10^7	5×10^2	1×10^4	1.03	17.2	1.09	969
2.2×10^7	5×10^2	2×10^4	1.03	4.21	1.02	2.58
2.2×10^7	5×10^2	5×10^4	1.05	15.5	0.97	725
2.2×10^7	1×10^3	1×10^4	0.934	4.91	0.961	16.2
2.2×10^7	1×10^3	2×10^4	0.686	3.39	0.875	3.52
2.2×10^7	1×10^3	5×10^4	1.05	20.6	1.04	7.09
2.2×10^7	2×10^3	1×10^4	0.789	14.8	0.89	65
2.2×10^7	2×10^3	2×10^4	0.914	15.1	0.91	5.27
2.2×10^7	2×10^3	5×10^4	1.06	10.6	1.1	5.45
3.3×10^7	5×10^2	1×10^4	1.33	11.3	0.929	772
3.3×10^7	5×10^2	2×10^4	0.923	15.9	1.05	20.2
3.3×10^7	5×10^2	5×10^4	0.935	5.44	1.06	12.2
3.3×10^7	1×10^3	1×10^4	1.33	7.02	1.09	22.6
3.3×10^7	1×10^3	2×10^4	1.02	6.49	0.973	4.27
3.3×10^7	1×10^3	5×10^4	1.03	7.98	0.962	31.3
3.3×10^7	2×10^3	1×10^4	1.12	3.16	0.992	26.4
3.3×10^7	2×10^3	2×10^4	0.918	4.51	0.94	23.2
3.3×10^7	2×10^3	5×10^4	0.9	12.3	1.05	6.98

Table F.3: Computed values of the fractal dimension β of particles generated by model B with different crystal prototype length. Parameters not listed in the table are those found in table 4.3. The values were computed following the procedure in section 4.5.1.

a	β_{cloud}	$\Pr(\beta_{\text{cloud}})$	β_{dropped}	$\Pr(\beta_{\text{dropped}})$
0.17	2.11	45.2	2.03	34.6
0.22	2.01	11.1	1.95	34.8
0.28	2.08	8.2	1.99	97.8
0.33	1.99	6.64	1.93	12
0.39	2.01	3.12	2	5.62
0.44	2.01	7.03	1.95	5.54
0.5	1.99	9.86	1.9	33.7
0.55	2	57.1	1.96	7.07
0.61	1.97	16.1	1.9	32.6
0.66	2.01	12.2	1.94	22.5

Table F.4: Computed values of the normalised volume growth coefficient α of particles generated by model B with different crystal prototype length. Parameters not listed in the table are those found in table 4.3. The values were computed following the procedure in section 4.5.1.

a	α_{cloud}	$\Pr(\alpha_{\text{cloud}})$	α_{dropped}	$\Pr(\alpha_{\text{dropped}})$
0.17	0.0635	4.65	0.0717	12.3
0.22	0.246	2.17	0.268	13.9
0.28	0.565	2.68	0.64	635
0.33	0.997	3	1.09	24
0.39	1.19	3.66	1.2	3.12
0.44	1.2	3.17	1.33	5.69
0.5	1.21	7.43	1.37	59.7
0.55	1.06	40.3	1.09	6.61
0.61	0.95	18.4	1.07	205
0.66	0.772	8.48	0.848	43

Table F.5: Computed values of the size distribution parameter γ among particles generated by model B. Parameters not listed in the table are those found in table 4.2. The values were computed following the procedure in section 4.5.1.

R_g	R_m	R_d	γ_{cloud}	$\Pr(\gamma_{\text{cloud}})$	γ_{dropped}	$\Pr(\gamma_{\text{dropped}})$
1.5×10^7	5×10^2	1×10^4	-0.382	328	-0.436	20.1
1.5×10^7	5×10^2	2×10^4	-0.581	376	-0.658	24.3
1.5×10^7	5×10^2	5×10^4	-1.22	69.9	-1.3	58.9
1.5×10^7	1×10^3	1×10^4	-0.456	71.3	-0.552	400
1.5×10^7	1×10^3	2×10^4	-0.685	36.8	-0.8	30.8
1.5×10^7	1×10^3	5×10^4	-1.48	20.8	-1.56	21.3
1.5×10^7	2×10^3	1×10^4	-0.798	42.6	-0.915	75.7
1.5×10^7	2×10^3	2×10^4	-1.22	69.5	-1.28	33.6
1.5×10^7	2×10^3	5×10^4	-2.04	8.15	-2.08	12.2
2.2×10^7	5×10^2	1×10^4	-0.316	609	-0.401	15.3
2.2×10^7	5×10^2	2×10^4	-0.491	83.3	-0.567	24.6
2.2×10^7	5×10^2	5×10^4	-0.956	115	-1.02	61.6
2.2×10^7	1×10^3	1×10^4	-0.352	22.8	-0.47	12.5
2.2×10^7	1×10^3	2×10^4	-0.574	157	-0.674	57.8
2.2×10^7	1×10^3	5×10^4	-1.11	62.1	-1.23	39.3
2.2×10^7	2×10^3	1×10^4	-0.539	31.5	-0.673	7.67
2.2×10^7	2×10^3	2×10^4	-0.837	182	-1.01	19.4
2.2×10^7	2×10^3	5×10^4	-1.48	20.9	-1.72	22.7
3.3×10^7	5×10^2	1×10^4	-0.277	45.6	-0.385	11.8
3.3×10^7	5×10^2	2×10^4	-0.385	85.4	-0.484	34.3
3.3×10^7	5×10^2	5×10^4	-0.718	70.3	-0.829	47.4
3.3×10^7	1×10^3	1×10^4	-0.301	106	-0.408	10.3
3.3×10^7	1×10^3	2×10^4	-0.459	68.4	-0.555	21.4
3.3×10^7	1×10^3	5×10^4	-0.846	28.9	-0.958	10.4
3.3×10^7	2×10^3	1×10^4	-0.42	34.4	-0.598	5.97
3.3×10^7	2×10^3	2×10^4	-0.615	55.3	-0.762	7.15
3.3×10^7	2×10^3	5×10^4	-1.11	184	-1.22	58.2

Table F.6: Computed values of the size distribution parameter γ among particles generated by model B with crystal prototype length. Parameters not listed in the table are those found in table 4.3. The values were computed following the procedure in section 4.5.1.

a	γ_{cloud}	$\Pr(\gamma_{\text{cloud}})$	γ_{dropped}	$\Pr(\gamma_{\text{dropped}})$
0.17	-0.43	29.2	-0.471	23.2
0.22	-0.451	18.5	-0.53	90.8
0.28	-0.439	429	-0.51	16.5
0.33	-0.398	37.4	-0.479	11.8
0.39	-0.375	598	-0.46	14.1
0.44	-0.357	51.8	-0.467	50.2
0.5	-0.376	69.4	-0.485	43.2
0.55	-0.408	75.6	-0.497	96
0.61	-0.417	41.2	-0.521	27
0.66	-0.451	50.1	-0.524	26.9

Table F.7: Computed values of the mean spherical volume fill ratio of particles generated by model B. Parameters not listed in the table are those found in table 4.2. The values were computed following the procedure in section 4.5.1.

R_g	R_m	R_d	$v_{r0\text{cloud}}$	$\Pr(v_{r0\text{cloud}})$	$v_{r0\text{dropped}}$	$\Pr(v_{r0\text{dropped}})$
1.5×10^7	5×10^2	1×10^4	0.0281	30.6	0.0311	72.9
1.5×10^7	5×10^2	2×10^4	0.038	14.7	0.0422	20.1
1.5×10^7	5×10^2	5×10^4	0.0655	15	0.0691	20.6
1.5×10^7	1×10^3	1×10^4	0.0293	33.3	0.0358	8.08
1.5×10^7	1×10^3	2×10^4	0.0418	40.8	0.0502	9.78
1.5×10^7	1×10^3	5×10^4	0.0754	41.8	0.0784	22.9
1.5×10^7	2×10^3	1×10^4	0.0469	34.8	0.053	22.8
1.5×10^7	2×10^3	2×10^4	0.0647	8.5	0.0673	13.6
1.5×10^7	2×10^3	5×10^4	0.0913	20.4	0.091	20.4
2.2×10^7	5×10^2	1×10^4	0.025	50.8	0.0285	37
2.2×10^7	5×10^2	2×10^4	0.0317	66.4	0.0359	83.2
2.2×10^7	5×10^2	5×10^4	0.0551	43.4	0.059	35.6
2.2×10^7	1×10^3	1×10^4	0.0288	107	0.0337	4.84
2.2×10^7	1×10^3	2×10^4	0.0371	23.9	0.0422	15.4
2.2×10^7	1×10^3	5×10^4	0.06	13.8	0.0677	19.6
2.2×10^7	2×10^3	1×10^4	0.0362	8.68	0.0461	15.4
2.2×10^7	2×10^3	2×10^4	0.0484	18.3	0.0565	64
2.2×10^7	2×10^3	5×10^4	0.0769	13.5	0.0841	6.6
3.3×10^7	5×10^2	1×10^4	0.0238	19.4	0.0292	8.9
3.3×10^7	5×10^2	2×10^4	0.0285	24.2	0.0321	17.5
3.3×10^7	5×10^2	5×10^4	0.0436	26.5	0.0492	82.3
3.3×10^7	1×10^3	1×10^4	0.0251	12.8	0.031	11.7
3.3×10^7	1×10^3	2×10^4	0.0308	10.8	0.0367	6.98
3.3×10^7	1×10^3	5×10^4	0.0498	14.3	0.0567	3.76
3.3×10^7	2×10^3	1×10^4	0.0316	9.21	0.0382	15.7
3.3×10^7	2×10^3	2×10^4	0.0366	7.1	0.0477	3.65
3.3×10^7	2×10^3	5×10^4	0.0621	22.7	0.0686	11.1

Table F.8: Computed values of the mean spherical volume fill ratio of particles generated by model B. Parameters not listed in the table are those found in table 4.3. The values were computed following the procedure in section 4.5.1.

a	$v_{r0\text{cloud}}$	$\Pr(v_{r0\text{cloud}})$	$v_{r0\text{dropped}}$	$\Pr(v_{r0\text{dropped}})$
0.17	0.00224	4.72	0.00266	7.53
0.22	0.00879	66.9	0.00964	12.7
0.28	0.0216	16.5	0.0241	6.29
0.33	0.0297	68.4	0.0318	8.53
0.39	0.0323	35.7	0.0404	33.9
0.44	0.0338	63.3	0.0399	69.3
0.5	0.0317	75	0.0376	6.62
0.55	0.0316	56.6	0.0364	10.6
0.61	0.0284	25.4	0.0321	6.71
0.66	0.0271	62	0.0293	8.23