

BornAgain

Software for simulating and fitting
X-ray and neutron small-angle scattering
at grazing incidence

User Manual

0.2.2

March 27, 2014

C. Durniak, M. Ganeva, G. Pospelov, W. Van Herck, J. Wuttke

Scientific Computing Group

Jülich Centre for Neutron Science

outstation at Heinz Maier-Leibnitz Zentrum Garching

Forschungszentrum Jülich GmbH

Disclaimer

This manual is under development and does not yet constitute a comprehensive listing of BornAgain features and functionality. The included information and instructions are subject to substantial changes and are provided only as a preview.

Contents

Introduction	4
1 Quick start	6
1.1 Quick start on Unix Platforms	6
1.2 Quick start on Windows Platforms	7
1.3 Getting help	7
2 Installation	8
2.1 Building and installing on Unix Platforms	8
2.1.1 Third-party software	9
2.1.2 Getting BornAgain source code	10
2.1.3 Building and installing the code	11
2.1.4 Running the first simulation	12
2.2 Installing on Windows Platforms	12
3 Simulation	14
3.1 General methodology	14
3.2 Geometry of the sample	14
3.3 Example 1: two types of islands on top of substrate without interference	16
3.4 Example 2: working with sample parameters	20
4 Fitting	24
4.1 Implementation in BornAgain	24
4.1.1 Preparing the sample and the simulation description	26
4.1.2 Choice of parameters to be fitted	26
4.1.3 Associating reference and simulated data	27
4.1.4 Minimizer settings	27
4.1.5 Running the fitting and retrieving the results	29
4.2 Basic Python fitting example	29
4.3 Advanced fitting	33
4.3.1 Affecting χ^2 calculations	33
4.3.2 Simultaneous fits of several data sets	33
4.3.3 Using fitting strategies	33
4.3.4 Masking the real data	33

4.3.5	Tuning fitting algorithms	33
4.3.6	Fitting with correlated sample parameters	33
4.4	How to get the right answer from fitting	33
5	Software architecture	35
5.1	Data classes for simulations and fits	36
5.1.1	The Experiment object	36
5.1.2	The ISample class hierarchy	37
5.1.3	The FitSuite class	38
5.1.4	The IMinimizer class	38
5.1.5	The MinimizerOptions class	38
A	Listings	39
A.1	Python simulation example	39
A.2	Python fitting example	41
B	Form factors	43
B.1	Box	46
B.2	Prism3	48
B.3	Tetrahedron	50
B.4	Prism6	52
B.5	Cone6	54
B.6	Pyramid	56
B.7	Anisotropic pyramid	58
B.8	Cuboctahedron	60
B.9	Cylinder	62
B.10	Ellipsoidal cylinder	64
B.11	Cone	66
B.12	Full Sphere	68
B.13	Truncated Sphere	70
B.14	Full Spheroid	72
B.15	Truncated Spheroid	74
B.16	Hemi ellipsoid	76
B.17	Ripple1	78
B.18	Ripple2	80
B.19	Distorted Wave Born Approximation	82
B.20	Core-shell particles	90
C	Collection of particles	93
C.1	Theory	93
C.2	Implementation in BornAgain	96
C.3	Summary	111

Introduction

BornAgain is a free software package to simulate and fit small-angle scattering at grazing incidence (GISAS). It supports analysis of both X-ray (GISAXS) and neutron (GISANS) data. Its name, BornAgain, indicates the central role of the distorted-wave Born approximation (DWBA) in the physical description of the scattering process. The software provides a generic framework for modeling multilayer samples with smooth or rough interfaces and with various types of embedded nanoparticles.

BornAgain almost completely reproduces the functionality of the widely used program IsGISAXS by R. Lazzari [1].

BornAgain goes beyond IsGISAXS by supporting an unrestricted number of layers and particles, diffuse reflection from rough layer interfaces, particles with inner structures, neutron polarization and magnetic scattering. Adhering to a strict object-oriented design, BornAgain provides a solid base for future extensions in response to specific user needs.

BornAgain is a platform-independent software, with active support for Linux, MacOS and Microsoft Windows. It is a free and open source software provided under the terms of the GNU General Public License (GPL). This documentation is released under the Creative Commons license CC-BY-SA.

The authors will be grateful for all kind of feedback: criticism, praise, bug reports, feature requests or contributed modules. When BornAgain is used in preparing scientific papers, please cite this manual as follows:

C. Durniak, M. Ganeva, G. Pospelov, W. Van Herck, J. Wuttke (2013),
BornAgain - Software for simulating and fitting X-ray and neutron small-angle
scattering at grazing incidence, version 0.2.2,
<http://apps.jcns.fz-juelich.de/BornAgain>

This user guide starts with a brief description of the steps necessary for installing the software and running a simulation on Unix and Windows platforms in Section 1. A more detailed description of the installation procedure is given in Section 2. The general methodology of a simulation with BornAgain and detailed simulation usage examples are given in Section 3. The fitting toolkit, provided by the framework, is presented in Section 4, while

Section 5 provides a brief overview of the software architecture.

Icons used in this manual:

 : this sign highlights further remarks.

 : this sign highlights essential points.

Chapter 1

Quick start

1.1 Quick start on Unix Platforms

This section shortly describes how to build and install BornAgain from source and run the first simulation on Unix Platforms. Further details about the installation procedure are given in Section 2.

Step I: installing the third party software

- compilers: clang versions ≥ 3.1 or GCC versions ≥ 4.2
- cmake (≥ 2.8)
- boost library (≥ 1.48)
- GNU scientific library (≥ 1.15)
- fftw3 library ($\geq 3.3.1$)
- Python-2.7, python-devel, python-numpy-devel

Step II: getting the source

Download BornAgain source tarball from <http://apps.jcns.fz-juelich.de/BornAgain> or use the following git repository

```
git clone git://apps.jcns.fz-juelich.de/BornAgain.git
```

Step III: building the libraries and executable

```
mkdir <build_dir>; cd <build_dir>;  
cmake -DCMAKE_INSTALL_PREFIX=<install_dir> <source_dir>  
make  
make check  
make install
```

Step IV: running an example

```
python <install_dir>/share/BornAgain/Examples/python/simulation/  
ex001_CylindersAndPrisms/CylindersAndPrisms.py
```

1.2 Quick start on Windows Platforms

Step I: installing the third party software

The current version of BornAgain requires Python, numpy, matplotlib to be installed on the system. If you don't have them already installed, you can use PythonXY installer available at <https://code.google.com/p/pythonxy> which, with default installation options, contains at least these three packages.

Step II: using BornAgain installation package

Windows installation package can be downloaded from <http://apps.jcns.fz-juelich.de/BornAgain>. Double-click on it to start the installation process. Then follow the instructions.

Step III: running the example

Run an example located in BornAgain installation directory:

```
python C:/BornAgain-0.9.4/Examples/python/simulation/  
ex001_CylindersAndPrisms/CylindersAndPrisms.py
```

1.3 Getting help

Users of the software who encounter problems during the installation of the framework or during the run of a simulation can use the web-based issue tracking system at <http://apps.jcns.fz-juelich.de/redmine/projects/bornagain/issues> to report a bug. The same system can be used to request new features. This system is open for all users in read mode, while submitting bug reports and feature requests are possible only after a simple registration procedure.

Chapter 2

Installation

BornAgain is supported under x86/x86_64 Linux, Mac OS X and Windows operating systems. It has been successfully compiled and tested on

- Microsoft Windows 7 64-bit, Windows 8 64-bit
- Mac OS X 10.8 (Mountain Lion), 10.9 (Maverick)
- OpenSuse 12.3 64-bit
- Ubuntu 12.10, 13.04 64-bit
- Debian 7.1.0, 32-bit, 64-bit

At the moment we support build and installation from source on Unix Platforms (Linux, Mac OS) and installation using binary installer packages on MS Windows 7, 8 (see Section 2.1 and Section 2.2, respectively). In the next releases we are planning to provide binary installers for Mac OS X and Debian.

We welcome feedback and bug reports related to installation and use of BornAgain via <http://apps.jcns.fz-juelich.de/redmine/projects/bornagain/issues>

2.1 Building and installing on Unix Platforms

BornAgain uses CMake to configure a build system for compiling and installing the framework. There are three major steps to build BornAgain :

1. Acquiring the required third-party libraries.
2. Getting BornAgain source code.
3. Using CMake to build and install the software.

The remainder of this section explains each step in detail.

2.1.1 Third-party software

To successfully build BornAgain a number of prerequisite packages must be installed.

- compilers: clang versions ≥ 3.1 or GCC versions $\geq 4.1.2$
- cmake ($\geq 2.8.3$)
- boost library (≥ 1.48)
- GNU scientific library (≥ 1.15)
- fftw3 library (≥ 3.3)
- Python (≥ 2.7 , < 3.0), python-devel, python-numpy-devel

Other packages are optional

- ROOT framework (adds several additional fitting algorithms to BornAgain)
- python-matplotlib (allows to run usage examples with graphics)

All required packages can be easily installed on most Linux distributions using the system's package manager. Below we give examples for a few selected operation systems. Please note, that other distributions (Fedora, Mint, etc) may have different commands for invoking the package manager as well as slightly different names of packages (like "boost" instead of "libboost" etc). Besides the installation should be very similar.

Ubuntu (12.10, 13.04), Debian (7.1)

Installing the required packages

```
sudo apt-get install git cmake libgsl0-dev libboost-all-dev  
libfftw3-dev python-dev python-numpy
```

Installing the optional packages

```
sudo apt-get install libroot-* root-plugin-* root-system-* ttf-  
root-installer libeigen3-dev python-matplotlib python-  
matplotlib-tk
```

OpenSuse 12.3

Adding the "scientific" repository

```
sudo zypper ar http://download.opensuse.org/repositories/science/  
openSUSE_12.3 science
```

Installing the required packages

```
sudo zypper install git-core cmake gsl-devel boost-devel fftw3-  
devel python-devel python-numpy-devel
```

Installing the optional packages

```
sudo zypper install libroot-* root-plugin-* root-system-* root-  
ttf libeigen3-devel python-matplotlib
```

Mac OS X 10.8, 10.9

To simplify the installation of third party open-source software on a Mac OS X system we recommend the use of MacPorts package manager. The easiest way to install MacPorts is by downloading the dmg from www.macports.org/install.php and running the system's installer. After the installation new command "port" will be available in a terminal window of your Mac.

Installing the required packages

```
sudo port -v selfupdate  
sudo port install git-core cmake  
sudo port install fftw-3 gsl  
sudo port install boost -no_single-no_static+python27  
sudo port select --set python python27
```

Installing the optional packages

```
sudo port install py27-matplotlib py27-numpy py27-scipy  
sudo port install root +fftw3+python27  
sudo port install eigen3
```

2.1.2 Getting BornAgain source code

BornAgain source can be downloaded at <http://apps.jcns.fz-juelich.de/BornAgain> and unpacked with

```
tar xzf bornagain-<version>.tar.gz
```

Alternatively one can obtain BornAgain source from our public Git repository.

```
git clone git://apps.jcns.fz-juelich.de/BornAgain.git
```

More about Git

Our Git repository holds two main branches called "master" and "develop". We consider "master" branch to be the main branch where the source code of HEAD always reflects the latest stable release. git clone command shown above

1. gives you a source code snapshot corresponding to the latest stable release,
2. automatically sets up your local master branch to track our remote master branch, so you will be able to fetch changes from the remote branch at any time using git pull command.

“Master” branch is updated approximately once per month. The second branch, “develop” branch, is a snapshot of the current development. This is where any automatic nightly builds are built from. The develop branch is always expected to work. So in order to get the most recent features of the source code, one can switch to it by

```
cd BornAgain
git checkout develop
git pull
```


2.1.3 Building and installing the code

BornAgain should be built using CMake cross platform build system. Having the third-party libraries installed on your system and BornAgain source code acquired as explained in the previous sections, type the build commands

```
mkdir <build_dir>
cd <build_dir>
cmake -DCMAKE_INSTALL_PREFIX=<install_dir> <source_dir>
make
```

Here <source_dir> is the name of the directory, where BornAgain source code has been copied, <install_dir> is the directory, where you want the package to be installed, and <build_dir> is the directory where the building will occur.

About CMake

 Having a dedicated directory <build_dir> for the build process is recommended by CMake. This allows several builds with different compilers/options from the same source and keeps the source directory clean from build remnants.

The compilation process invoked by the command “make” lasts about 10 minutes on an average laptop of 2012 edition. On multi-core machines the compilation time can be decreased by invoking command “make” with the parameter “make -j[N]”, where N is the number of cores.

Running functional tests is an optional but recommended step. Command “make check” will compile several additional tests and run them one by one. Each test contains the simulation of a typical GISAS geometry and the comparison on numerical level of simulation results with reference files. Having 100% tests passed ensures that your local installation is correct.

```
make check
...
100% tests passed, 0 tests failed out of 26
Total Test time (real) = 89.19 sec
[100%] Build target check
```


The last command “make install” copies the compiled libraries and some usage examples into the installation directory.

```
make install
```

After installation is completed, the location of BornAgain libraries needs to be included into LD_LIBRARY_PATH and PYTHONPATH environment variables. This can be done by running BornAgain setup script in the terminal session

```
source <install_dir>/bin/thisbornagain.sh
```

Conveniently, given call can be placed in your .bashrc file.

Troubleshooting

In the case of a complex system setup, with libraries of different versions scattered across multiple places (/opt/local, /usr/local etc.), you may want to help CMake in finding the correct library paths by running cmake with additional parameter

```
cmake -DCMAKE_PREFIX_PATH=/usr/local -DCMAKE_INSTALL_PREFIX=<
install_dir> <source_dir>
```

2.1.4 Running the first simulation

In your installation directory you will find

```
./include/BornAgain - header files for compilation of your C++
program
./lib - libraries to import into python or link with your C++
program
./share/BornAgain/Examples - directory with examples
```

Run your first example and enjoy the first BornAgain simulation plot.

```
python <install_dir>/share/BornAgain/Examples/python/simulation/
ex001_CylindersAndPrisms/CylindersAndPrisms.py
```

2.2 Installing on Windows Platforms

Step I: install the third party software

The current version of BornAgain requires Python, numpy, matplotlib to be installed on the system.

If you do not have have Python installed

You can use PythonXY installer at <https://code.google.com/p/pythonxy> which, with the default installation options, contains at least these three packages. The user has to download and install this package before proceeding to the installation of BornAgain.

If you have Python already installed

You might want to keep using this installation and to install missed modules. Required libraries can be found at

```
matlab:  
http://matplotlib.org/downloads.html  
  
numpy, dateutil, pyparsing:  
http://www.lfd.uci.edu/~gohlke/pythonlibs
```

Step II: use the installation package

BornAgain installation package for Windows can be downloaded from <http://apps.jcns.fz-juelich.de/BornAgain>. Double-click on it to start the installation process. And then follow the instructions.

Step IV: run an example

Run an example located in BornAgain installation directory:

```
python C:/BornAgain-0.9.4/Examples/python/simulation/  
ex001_CylindersAndPrisms/CylindersAndPrisms.py
```

Chapter 3

Simulation

3.1 General methodology

A simulation of GISAXS using BornAgain consists of following steps:

- define materials by specifying name and refractive index,
- define embedded particles by specifying shape, size, constituting material, interference function,
- define layers by specifying thickness, roughness, material,
- include particles in layers, specifying density, position, orientation,
- assemble a multilayered sample,
- specify input beam and detector characteristics,
- run the simulation,
- save the simulated detector image.

We are planing to organize all these steps in a graphical user interface (GUI). For the time being, however, BornAgain must be involved via C++ program or Python scripts. In the following, we describe how to write a Python script which runs a BornAgain simulation. For tutorials about this programming language, the users are referred to [2].

More information about the general software architecture and BornAgain internal design are given in Section 5.

3.2 Geometry of the sample

The geometry used to describe the sample is shown in figure 3.1. The z -axis is perpendicular to the sample's surface and pointing upwards. The x -axis is perpendicular to the detector plane. The input and the scattered output beams are each characterized by two

angles α_i , ϕ_i and α_f , ϕ_f , respectively. Our choice of orientation for the angles α_i and α_f is so that they are positive as shown in figure 3.1.

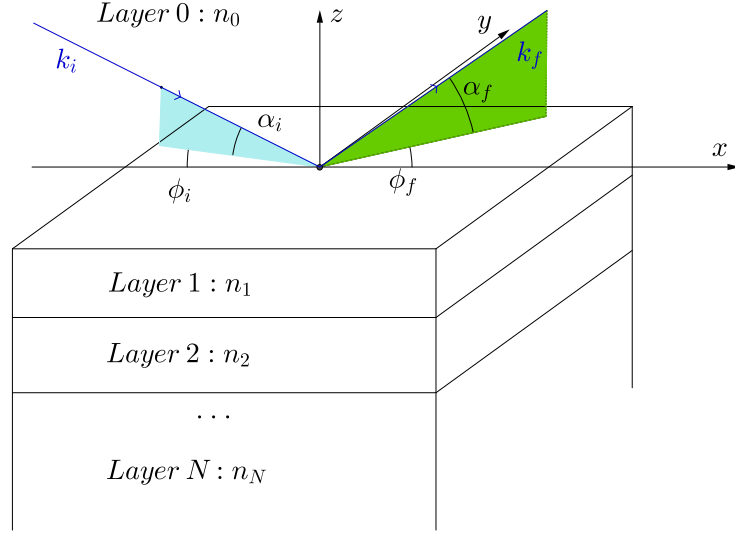


Figure 3.1: Representation of the scattering geometry. n_j is the refractive index of layer j and α_i and ϕ_i are the incident angles of the wave propagating. α_f is the exit angle with respect to the sample's surface and ϕ_f is the scattering angle with respect to the scattering plane.

The layers are defined by their thicknesses (parallel to the z -direction), their possible roughnesses (equal to 0 by default) and the material they are made of. They have infinite extension in the x , y directions. And, except for roughness, they interfaces are plane and perpendicular to the z -axis. There is also no limitation to the number of layers that could be defined in BornAgain. Note that the thickness of the top and bottom layer are not defined.

The nanoparticles are characterized by their form factors (*i.e.* the Fourier transform of the shape function - see Appendix B for a list of form factors implemented in BornAgain) and the composing material. The number of input parameters for the form factor depends on the particle symmetry; it ranges from one parameter for a sphere (its radius) to three for an ellipsoid (its three main axis lengths).

By placing the particles inside or on top of a layer, we impose their vertical positions, whose values correspond to the bottoms of the particles. The in-plane distribution of particles is linked with the way the particles interfere with each other. It is therefore implemented when dealing with the interference function.

The complex refractive index associated with a layer or a particle is written as $n = 1 - \delta + i\beta$, with $\delta, \beta \in \mathbb{R}$. In our program, we input δ and β directly.

The input beam is assumed to be monochromatic without any spatial divergence.

Units: By default the angles are expressed in radians and the lengths are given in nanometers. But it is possible to use other units by specifying them right after the value of the corresponding parameter like, for example, `20.0*micrometer`.

3.3 Example 1: two types of islands on top of substrate without interference

In this example, we simulate the scattering from a mixture of cylindrical and prismatic nanoparticles without any interference between them. These particles are placed in air, on top of a substrate.

We are going to go through each step of the simulation. The Python script specific to each stage will be given at the beginning of the description. But for the sake of completeness the full code is given in Appendix A.1.

Importing Python modules

```
1 import numpy
2 import matplotlib
3 import pylab
4 from libBornAgainCore import *
```

We start by importing different functions from external modules, for example NumPy (lines 1-3), which is a fundamental package for scientific computing with Python [3]. In particular, line 4 imports the features of BornAgain software.

Defining the materials

```
5 def get_sample():
6     """
7     Build and return the sample representing cylinders and
8     pyramids on top of
9     substrate without interference.
10    """
11    # defining materials
12    m_air = MaterialManager.getHomogeneousMaterial("Air", 0.0,
13    0.0)
14    m_substrate = MaterialManager.getHomogeneousMaterial("
15    Substrate", 6e-6, 2e-8)
16    m_particle = MaterialManager.getHomogeneousMaterial("Particle
17    ", 6e-4, 2e-8)
```

Line 5 marks the beginning of the function to define our sample. Lines 11, 12 and 13 define different materials using function `getHomogeneousMaterial` from class `MaterialManager`. The general syntax is the following

```
<material_name> = MaterialManager.getHomogeneousMaterial("name",
    delta, beta)
```

where name is the name of the material associated with its complex refractive index $n=1-\delta+i\beta$. <material_name> is later used when referring to this particular material. The three defined materials in this example are Air with a refractive index of 1 ($\delta = \beta = 0$), a Substrate associated with a complex refractive index equal to $1 - 6 \times 10^{-6} + i2 \times 10^{-8}$, and the material of particles, whose refractive index is $n = 1 - 6 \times 10^{-4} + i2 \times 10^{-8}$.

Defining the particles

```
15 # collection of particles
16 cylinder_ff = FormFactorCylinder(5*nanometer, 5*nanometer)
17 cylinder = Particle(m_particle, cylinder_ff)
18 prism_ff = FormFactorPrism3(10*nanometer, 5*nanometer)
19 prism = Particle(m_particle, prism_ff)
```

We implement two different shapes of particles: cylinders and prisms (*i.e.* elongated particles with a constant equilateral triangular cross section).

All particles implemented in BornAgain are defined by their form factors (see Appendix B), their sizes and the material they are made of. Here, for the cylindrical particle, we input its radius and height. For the prism, the possible inputs are the length of one side of its equilateral triangular base and its height.

In order to define a particle, we proceed in two steps. For example for the cylindrical particle, we first specify the form factor of a cylinder with its radius and height, both equal to 5 nanometers in this particular case (see line 16). Then we associate this shape with the constituting material as in line 17. The same procedure has been applied for the prism in lines 18 and 19, respectively.

Characterizing particles assembly

```
20 particle_decoration = ParticleDecoration()
21 particle_decoration.addParticle(cylinder, 0.0, 0.5)
22 particle_decoration.addParticle(prism, 0.0, 0.5)
23 interference = InterferenceFunctionNone()
24 particle_decoration.addInterferenceFunction(interference)
```

The object which holds the information about the positions and densities of particles in our sample is called ParticleDecoration (line 20). We use the associated function addParticle for each particle shape (lines 21, 22). Its general syntax is

```
addParticle(<particle_name>, depth, abundance)
```

where <particle_name> is the name used to define the particles (lines 17 and 19), depth (default value =0) is the vertical position, expressed in nanometers, of the particles in a given layer (the association with a particular layer will be done during the next step) and

abundance is the proportion of this type of particles, normalized to the total number of particles. Here we have 50% of cylinders and 50% of prisms.

Remark: Depth of particles



The vertical positions of the particles in a layer are given in relative coordinates. For the top layer, the bottom of the layer corresponds to $\text{depth}=0$ and negative values would correspond to particles floating above layer 1 since the vertical axis, shown in figure 3.1 is pointing upwards. But for all the other layers, it is the top of the layer which corresponds to $\text{depth}=0$.

Finally, lines 23 and 24 specify that there is **no coherent interference** between the waves scattered by these particles. In this case, the intensity is calculated by the incoherent sum of the scattered waves: $\langle |F_j|^2 \rangle$, where F_j is the form factor associated with the particle of type j . The way these waves interfere imposes the horizontal distribution of the particles as the interference reflects the long or short-range order of the particles distribution (**see Theory**). On the contrary, the vertical position is imposed when we add the particles in a given layer by parameter `depth`, as shown in lines 21 and 22.

Multilayer

```

25 # air layer with particles and substrate form multi layer
26     air_layer = Layer(m_air)
27     air_layer.setDecoration(particle_decoration)
28     substrate_layer = Layer(m_substrate, 0)
29     multi_layer = MultiLayer()
30     multi_layer.addLayer(air_layer)
31     multi_layer.addLayer(substrate_layer)
32     return multi_layer

```

We now have to configure our sample. For this first example, the particles, *i.e.* cylinders and prisms, are on top of a substrate in an air layer. **The order in which we define these layers is important: we start from the top layer down to the bottom one.**

Let us start with the air layer. It contains the particles. In line 26, we use the previously defined `mAmbience` (`"air"` material) (line 11). The command in line 27 shows that this layer is decorated by adding the particles using particle decoration object defined earlier. The substrate layer only contains the substrate material (line 28).

There are different possible syntaxes to define a layer. As shown in lines 26 and 28, we can use `Layer(<material_name>, thickness)` or `Layer(<material_name>)`. The second case corresponds to the default value of the thickness, equal to 0. The thickness is expressed in nanometers.

Our two layers are now fully characterized. The sample is assembled using `MultiLayer()` constructor (line 29): we start with the air layer decorated with the particles (line 30), which is the layer at the top and end with the bottom layer, which is the substrate (line 31).

Characterizing the input beam and output detector

```

33 def get_simulation():
34     """
35     Create and return GISAXS simulation with beam and detector
       defined
36     """
37     simulation = Simulation()
38     simulation.setDetectorParameters(100, -1.0*degree, 1.0*degree
       , 100, 0.0*degree, 2.0*degree, True)
39     simulation.setBeamParameters(1.0*angstrom, 0.2*degree, 0.0*
       degree)
40     return simulation

```

The first stage is to create the `Simulation()` object (line 37). Then we define the detector (line 38) and beam parameters (line 39). Those functions are part of the `Simulation` class. The different incident and exit angles are shown in figure 3.1.

The detector parameters are set using ranges of angles via the function:

```

setDetectorParameters(n_phi, phi_f_min, phi_f_max, n_alpha,
    alpha_f_min, alpha_f_max, isgisaxs_style=False)},

```

where $n_phi=100$ is the number of iterations for ϕ_f ,

$phi_f_min=-1.0*degree$ and $phi_f_max=1.0*degree$ are the minimum and maximum values respectively of ϕ_f ,

$n_alpha=100$ is the number of iterations for α_f ,

$alpha_f_min=0.0*degree$ and $alpha_f_max=2.0*degree$ are the minimum and maximum values respectively of α_f .

`isgisaxs_style=True` (default value = `False`) is a boolean used to characterise the structure of the output data. If `isgisaxs_style=True`, the output data is binned at constant values of the sine of the output angles, α_f and ϕ_f , otherwise it is binned at constant values of these two angles.

For the beam the function to use is `setBeamParameters(lambda, alpha_i, phi_i)`, where $\lambda=1.0*angstrom$ is the incident beam wavelength, $\alpha_i=0.2*degree$ is the incident grazing angle on the surface of the sample, $\phi_i=0.0*degree$ is the in-plane direction of the incident beam (measured with respect to the x -axis).

Remark: Scattering vector



In BornAgain the wave vector \mathbf{q} is defined as $\mathbf{k}_i - \mathbf{k}_f$, where \mathbf{k}_i is the incident wave vector and \mathbf{k}_f the scattered one.

Running the simulation and plotting the results

```

41 def run_simulation():
42     """
43     Run simulation and plot results
44     """
45     sample = get_sample()

```



```

46     simulation = get_simulation()
47     simulation.setSample(sample)
48     simulation.runSimulation()
49     result = simulation.getIntensityData().getArray() + 1 # for
        log scale
50     pylab.imshow(numpy.rot90(result, 1), norm=matplotlib.colors.
        LogNorm(), extent=[-1.0, 1.0, 0, 2.0])
51     pylab.show()

```

The function, whose definition starts from line 41, gathers all items. We create the sample and the simulation objects at the lines 45 and 46, using calls to the previously defined functions. We assign the sample to the simulation at line 47 and finally launch the simulation at line 48.

In line 49 we obtain the simulated intensity as a function of outgoing angles α_f and ϕ_f for further uses (plots, fits,...) as a NumPy array containing $n_{\text{phi}} \times n_{\text{alpha}}$ datapoints. Lines 50-51 produces the two-dimensional contourplot of the intensity as a function of α_f and ϕ_f shown in figure 3.2.

3.4 Example 2: working with sample parameters

This section gives additional details about the manipulation of sample parameters during run time; that is after the sample has already been constructed. For a single simulation this is normally not necessary. However it might be useful during interactive work when the user tries to find optimal sample parameters by running a series of simulations. A similar task also arises when the theoretical model, composed of the description of the sample and of the simulation, is used for fitting real data. In this case, the fitting kernel requires a list of the existing sample parameters and a mechanism for changing the values of these parameters in order to find their optima.

In BornAgain this is done using the so-called sample parameter pool mechanism. We are going to briefly explain this approach using the example of Section 3.3.

In BornAgain a sample is described by a hierarchical tree of objects. For the multilayer created in the previous section this tree can be graphically represented as shown in Fig. 3.3. Similar trees can be printed in a Python session by running `multi_layer.printSampleTree()`

The top `MultiLayer` object is composed of three children, namely `Layer #0`, `Layer Interface #0` and `Layer #1`. The children objects might themselves also be decomposed into tree-like structures. For example, `Layer #0` contains a `ParticleDecoration` object, which holds information related to the two types of particles populating the layer. All numerical values used during the sample construction (thickness of layers, size of particles, roughness parameters) are part of the same tree structure. They are marked in the figure with shaded gray boxes.

These values are registered in the sample parameter pool using the name composed of the corresponding nodes' names. And they can be accessed/changed during run time. For example, the height of the cylinders populating the first layer can be changed from the current value of 5 nm to 1 nm by running the command

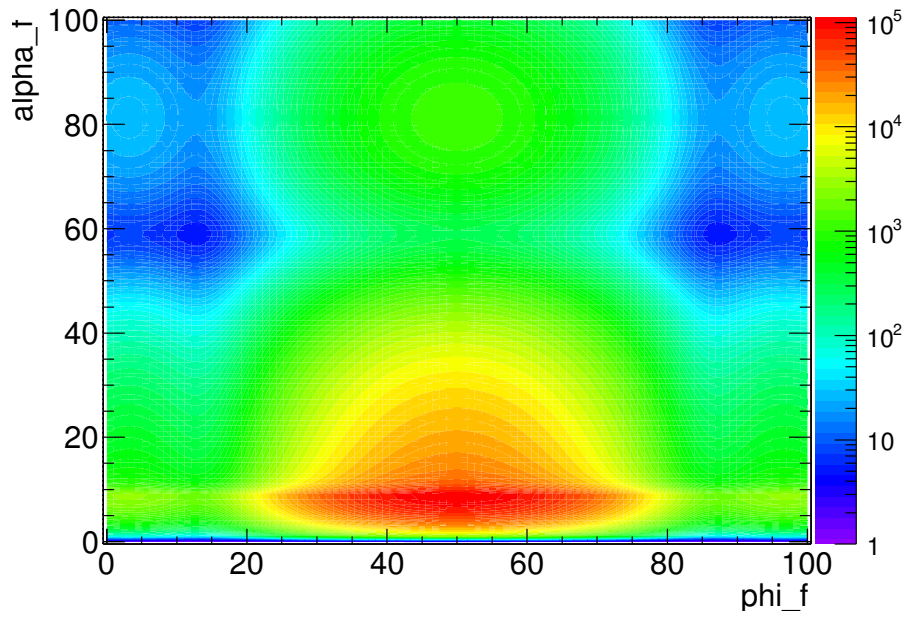


Figure 3.2: Simulated grazing-incidence small-angle X-ray scattering from a mixture of cylindrical and prismatic nanoparticles without any interference, deposited on top of a substrate. The input beam is characterized by a wavelength λ of 1 Å and incident angles $\alpha_i = 0.2^\circ$, $\phi_i = 0^\circ$. The cylinders have a radius and a height both equal to 5 nm, the prisms are characterized by a side length equal to 10 nm and they are 5 nm high. The material of the particles has a refractive index of $1 - 6 \times 10^{-4} + i2 \times 10^{-8}$. For the substrate it is equal to $1 - 6 \times 10^{-6} + i2 \times 10^{-8}$. The colorscale is associated with the output intensity in arbitrary units.

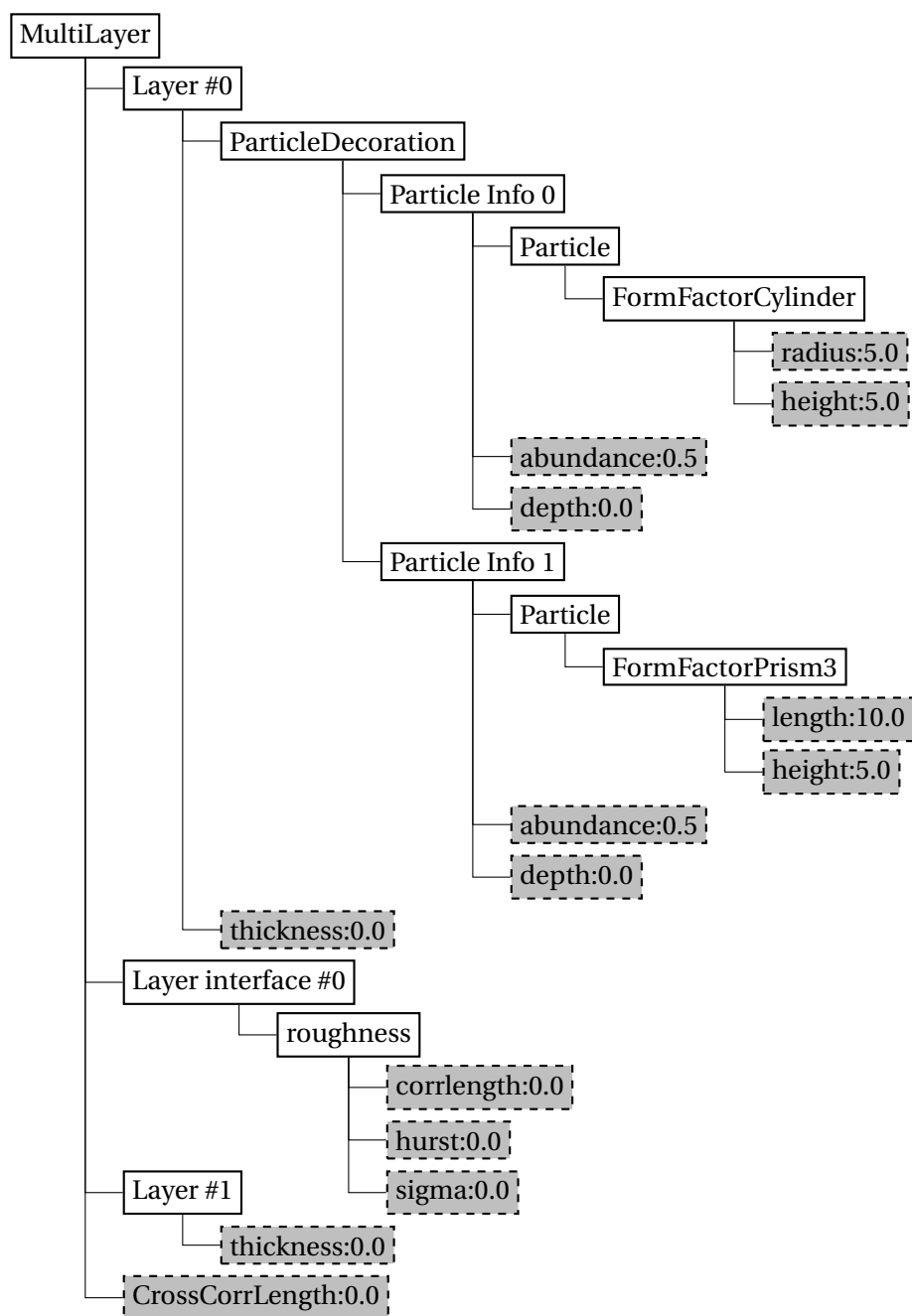


Figure 3.3: Tree representation of the sample structure.

```
multi_layer.setParameterValue('/MultiLayer/Layer0/
ParticleDecoration/ParticleInfo0/Particle/FormFactorCylinder/
height', 1.0)
```

A list of the names and values of all registered sample's parameters can be displayed using the command

```
> multi_layer.printParameters()
The sample contains following parameters ('name':value)
'/MultiLayer/Layer0/ParticleDecoration/ParticleInfo0/Particle/
FormFactorCylinder/height':5
'/MultiLayer/Layer0/ParticleDecoration/ParticleInfo0/Particle/
FormFactorCylinder/radius':5
'/MultiLayer/Layer0/ParticleDecoration/ParticleInfo0/abundance
':0.5
'/MultiLayer/Layer0/ParticleDecoration/ParticleInfo0/depth':0
'/MultiLayer/Layer0/ParticleDecoration/ParticleInfo1/Particle/
FormFactorPrism3/length':5
'/MultiLayer/Layer0/ParticleDecoration/ParticleInfo1/Particle/
FormFactorPrism3/height':5
'/MultiLayer/Layer0/ParticleDecoration/ParticleInfo1/abundance
':0.5
'/MultiLayer/Layer0/ParticleDecoration/ParticleInfo1/depth':0
'/MultiLayer/Layer0/thickness':0
'/MultiLayer/Layer1/thickness':0
'/MultiLayer/LayerInterface/roughness/corrlength':0
'/MultiLayer/LayerInterface/roughness/hurst':0
'/MultiLayer/LayerInterface/roughness/sigma':0
'/MultiLayer/crossCorrLength':0
```

Wildcards '*' can be used to reduce typing or to work on a group of parameters. In the example below, the first command will change the height of all cylinders in the same way, as in the previous example. The second line will change simultaneously the height of *both* cylinders and prisms.

```
multi_layer.setParameterValue('*FormFactorCylinder/height', 1.0)
multi_layer.setParameterValue('*height', 1.0)
```

The complete example described in this section can be found at

```
./Examples/python/fitting/ex001_SampleParametersIntro/
SampleParametersIntro.py
```

Chapter 4

Fitting

In addition to the simulation of grazing incidence X-ray and neutron scattering by multilayered samples, BornAgain also offers the option to fit the numerical model to reference data by modifying a selection of sample parameters from the numerical model. This aspect of the software is discussed in the current chapter.

Section 4.1 details the implementation of fittings in BornAgain . Python fitting examples with detailed explanations of every fitting step are given in Section 4.2. Advanced fitting techniques, including fine tuning of minimization algorithms, simultaneous fits of different data sets, parameters correlation, are covered in Section 4.3. Section 4.4 contains some practical advice, which might help the user to get right answers from BornAgain fitting.

4.1 Implementation in BornAgain

Fitting in BornAgain deals with estimating the optimum parameters in the numerical model by minimizing the difference between numerical and reference data. The features include

- a variety of multidimensional minimization algorithms and strategies.
- the choice over possible fitting parameters, their properties and correlations.
- the full control on objective function calculations, including applications of different normalizations and assignments of different masks and weights to different areas of reference data.
- the possibility to fit simultaneously an arbitrary number of data sets.

Figure 4.1 shows the general work flow of a typical fitting procedure.

Before running the fitting the user is required to prepare some data and to configure the fitting kernel of BornAgain . The required stages are

- Preparing the sample and the simulation description (multilayer, beam, detector parameters).

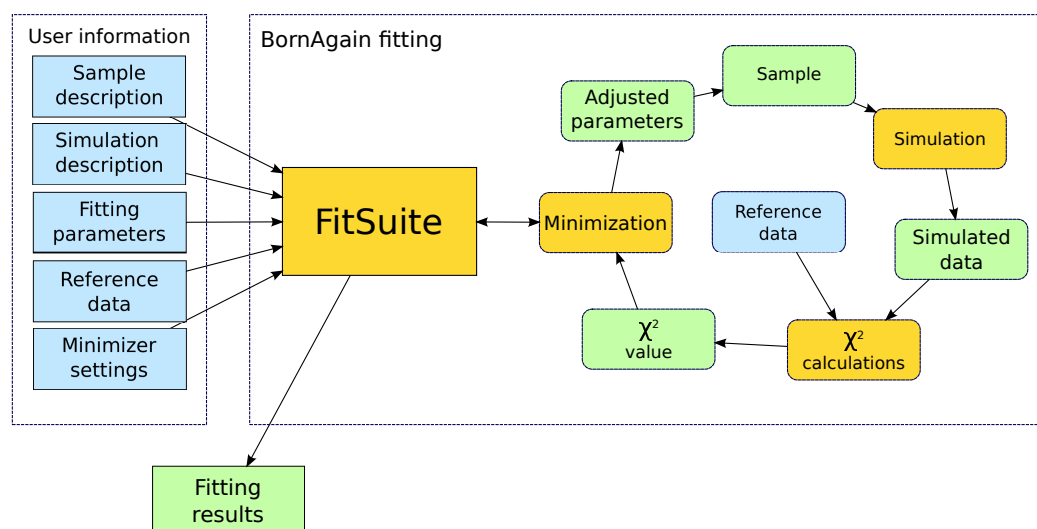


Figure 4.1: Fitting work flow.

- Choosing the fitting parameters.
- Loading the reference data.
- Defining the minimization settings.

The class `FitSuite` contains the main functionalities to be used for the fit and serves as the main interface between the user and the fitting work flow. The later involves iterations during which

- The minimizer makes an assumption about the optimal sample parameters.
- These parameters are propagated to the sample.
- The simulation is performed for the given state of the sample.
- The simulated data (intensities) are propagated to the χ^2 module.
- The later calculates χ^2 using the simulated and reference data.
- The value of χ^2 is propagated to the minimizer, which makes new assumptions about optimal sample parameters.

The iteration process is going on under the control of the selected minimization algorithm, without any intervention from the user. It stops

- when the maximum number of iteration steps has been exceeded,

- when the function's minimum has been reached within the tolerance window,
- if the minimizer could not improve the values of the parameters.

After the control is returned, fitting results can be retrieved. They consist in the best χ^2 value found, the corresponding optimal sample parameters and the intensity map simulated with this set of parameters.

Details of FitSuite class implementation and description of each interface are given in Section 5.1.3. The following parts of this section will detail each of the main stages necessary to run a fitting procedure.

4.1.1 Preparing the sample and the simulation description

This step is similar for any simulation using BornAgain (see Section 3). It consists in first characterizing the geometry of the system: the particles (shapes, sizes, refractive indices), the different layers (thickness, order, refractive index, a possible roughness of the interface), the interference between the particles and the way they are distributed in the layers (buried particles or particles sitting on top of a layer). Then we specify the parameters of the input beam and of the output detector.

4.1.2 Choice of parameters to be fitted

In principle, every parameter used in the construction of the sample can be used as a fitting parameter. For example, the particles' heights, radii or the layer's roughness or thickness could be selected using the parameter pool mechanism. This mechanism is explained in detail in Section 3.4 and it is therefore recommended to read it before proceeding any further.

The user specifies selected sample parameters as fit parameters using FitSuite and its addFitParameter method

```
fit_suite = FitSuite()
fit_suite.addFitParameter(<name>, <initial value>, <step>, <
    limits>)
```

where <name> corresponds to the parameter name in the sample's parameter pool. By using wildcards in the parameter name, a group of sample parameters, corresponding to the given pattern, can be associated with a single fitting parameter and fitted simultaneously to get a common optimal value (see Section 3.4).

The second parameter <initial value> correspond to the initial value of the fitting parameter, while the third one is responsible to the initial iteration steps size. The last parameter <AttLimits> corresponds to the boundaries imposed on parameter value. It can be

- limitless() by default,
- fixed(),
- lowerLimited(<min_value>),

- `upperLimited(<max_value>),`
- `limited(<min_value>, <max_value>).`

where `<min_value>` and `<max_value>` are double values corresponding to the lower and higher boundary, respectively.

4.1.3 Associating reference and simulated data

The minimization procedure deals with a pair of reference data (normally associated with experimental data) and the theoretical model (presented by the sample and the simulation descriptions).

We assume that the experimental data are a two-dimensional intensity matrix as function of the output scattering angles α_f and ϕ_f (see Fig. 3.1). The user is required to provide the data in the form of an ASCII file containing an axes binning description and the intensity data itself.



Remark: We recognize the importance of supporting the most common data formats. We are going to provide this feature in the following releases and welcome users' requests on this subject.

To associate the simulation and the reference data to the fitting engine, method `addSimulationAndRealData` has to be used as shown

```
fit_suite = FitSuite()
fit_suite.addSimulationAndRealData(<simulation>, <reference>, <
    chi2_module>)
```

Here `<simulation>` corresponds to a BornAgain simulation object with the sample, beam and detector fully defined, `<reference>` corresponds to the experimental data object obtained from the ASCII file and `<chi2_module>` is an optional parameter for advanced control of χ^2 calculations.

It is possible to call this given method more than once to submit more than one pair of `<simulation>`, `<reference>` to the fitting procedure. In this way, simultaneous fits of some combined data sets are performed.

By using the third parameter, `<chi2_module>`, different normalizations and weights can be applied to give user full control of the way χ^2 is calculated. This feature will be explained in Section 4.3.

4.1.4 Minimizer settings

BornAgain contains a variety of minimization engines from ROOT and GSL libraries. They are listed in Table 4.1. By default Minuit2 minimizer with default settings will be used and no additional configuration needs to be done. The remainder of this section explains some of the expert settings, which can be applied to get better fit results.

The default minimization algorithm can be changed using `MinimizerFactory` as shown below


```

fit_suite = FitSuite()
minimizer = MinimizerFactory.createMinimizer("<Minimizer name>", "<algorithm>")
fit_suite.setMinimizer(minimizer)

```

where <Minimizer name> and <algorithm> can be chosen from the first and second column of Table 4.1 respectively. The list of minimization algorithms implemented in BornAgain can also be obtained using `MinimizerFactory.printCatalogue()` command.

Minimizer name	Algorithm	Description
Minuit2 [4]	Migrad	According to [5] best minimizer for nearly all functions, variable-metric method with inexact line search, a stable metric updating scheme, and checks for positive-definiteness.
	Simplex	simplex method of Nelder and Mead usually slower than Migrad, rather robust with respect to gross fluctuations in the function value, gives no reliable information about parameter errors,
	Combined	minimization with Migrad but switches to Simplex if Migrad fails to converge.
	Scan	not intended to minimize, just scans the function, one parameter at a time, retains the best value after each scan
	Fumili	optimized method for least square and log likelihood minimizations
GSLMultiMin [6]	ConjugateFR	Fletcher-Reeves conjugate gradient algorithm,
	ConjugatePR	Polak-Ribiere conjugate gradient algorithm,
	BFGS	Broyden-Fletcher-Goldfarb-Shanno algorithm,
	BFGS2	improved version of BFGS,
	SteepestDescent	follows the downhill gradient of the function at each step
GSLLMMA [7]		Levenberg-Marquardt Algorithm
GSLSimAn [8]		Simulated Annealing Algorithm

Table 4.1: List of minimizers implemented in BornAgain.

There are several options common to every minimization algorithm, which can be changed before starting the minimization. They are handled by `MinimizerOptions` class:

```
fit_suite.getMinimizer().getOptions().setMaxFunctionCalls(10)
```

In the above code snippet, a number of “maximum function calls”, namely the maximum number of times the minimizer is allowed to call the simulation, is limited to 10.

There are also expert-level options common for all minimizers as well as a number of options to tune individual minimization algorithms. They will be explained in Section 4.3.

4.1.5 Running the fitting and retrieving the results

After the initial configuration of `FitSuite` has been performed, the fitting can be started using the command

```
fit_suite.runFit()
```

Depending on the complexity of the sample and the number of free sample parameters the fitting process can take from tens to thousands of iterations. The results of the fit can be printed on the screen using the command

```
fit_suite.printResults()
```

Section 4.2 gives more details about how to access the fitting results.

4.2 Basic Python fitting example

In this section we are going to go through a complete example of fitting using `BornAgain`. Each step will be associated with a detailed piece of code written in Python. The complete listing of the script is given in Appendix (see Listing A.2). The script can also be found at

```
./Examples/python/fitting/ex002_FitCylindersAndPrisms/  
FitCylindersAndPrisms.py
```

This example uses the same sample geometry as in Section 3.3. Cylindrical and prismatic particles in equal proportion are deposited on a substrate layer, with no interference between the particles. We consider the following parameters to be unknown

- the radius of cylinders,
- the height of cylinders,
- the length of the prisms’ triangular basis,
- the height of prisms.

Our reference data are a “noisy” two-dimensional intensity map obtained from the simulation of the same geometry with a fixed value of 5 nm for the height and radius of cylinders and for the height of prisms which have a 10-nanometer-long side length. Then we run our

fitting using default minimizer settings starting with a cylinder's height of 4 nm, a cylinder's radius of 6 nm, a prism's half side of 6 nm and a height equal to 4 nm. As a result, the fitting procedure is able to find the correct value of 5 nm for all four parameters.

Importing Python libraries

```
1 from libBornAgainCore import *
2 from libBornAgainFit import *
```

We start from importing two BornAgain libraries required to create the sample description and to run the fitting.

Building the sample

```
5 def get_sample():
6     """
7     Build the sample representing cylinders and pyramids on top
8     of substrate without interference.
9     """
10    # defining materials
11    m_air = MaterialManager.getHomogeneousMaterial("Air", 0.0,
12    0.0)
13    m_substrate = MaterialManager.getHomogeneousMaterial("
14    Substrate", 6e-6, 2e-8)
15    m_particle = MaterialManager.getHomogeneousMaterial("Particle
16    ", 6e-4, 2e-8)
17
18    # collection of particles
19    cylinder_ff = FormFactorCylinder(1.0*nanometer, 1.0*nanometer
20    )
21    cylinder = Particle(m_particle, cylinder_ff)
22    prism_ff = FormFactorPrism3(2.0*nanometer, 1.0*nanometer)
23    prism = Particle(m_particle, prism_ff)
24    particle_decoration = ParticleDecoration()
25    particle_decoration.addParticle(cylinder, 0.0, 0.5)
26    particle_decoration.addParticle(prism, 0.0, 0.5)
27    interference = InterferenceFunctionNone()
28    particle_decoration.addInterferenceFunction(interference)
29
30    # air layer with particles and substrate form multi layer
31    air_layer = Layer(m_air)
32    air_layer.setDecoration(particle_decoration)
33    substrate_layer = Layer(m_substrate)
34    multi_layer = MultiLayer()
35    multi_layer.addLayer(air_layer)
36    multi_layer.addLayer(substrate_layer)
37    return multi_layer
```

The function starting at line 5 creates a multilayered sample with cylinders and prisms using arbitrary 1 nm value for all size's of particles. The details about the generation of this multilayered sample are given in Section 3.3.

Creating the simulation

```
35 def get_simulation():
36     """
37     Create GISAXS simulation with beam and detector defined
38     """
39     simulation = Simulation()
40     simulation.setDetectorParameters(100, -1.0*degree, 1.0*degree
41     , 100, 0.0*degree, 2.0*degree, True)
42     simulation.setBeamParameters(1.0*angstrom, 0.2*degree, 0.0*
43     degree)
44     return simulation
```

The function starting at line 35 creates the simulation object with the definition of the beam and detector parameters.

Preparing the fitting pair

```
45 def run_fitting():
46     """
47     run fitting
48     """
49     sample = get_sample()
50     simulation = get_simulation()
51     simulation.setSample(sample)
52
53     real_data = OutputDataIOFactory.readIntensityData('
54         refdata_fitcylinderprisms.txt')
```

Lines 49- 51 generate the sample and simulation description and assign the sample to the simulation. Our reference data are contained in the file 'refdata_fitcylinderprisms.txt'. This reference had been generated by adding noise on the scattered intensity from a numerical sample with a fixed length of 5 nm for the four fitting parameters (*i.e.* the dimensions of the cylinders and prisms). Line 53 creates the real data object by loading the ASCII data from the input file.

Setting up FitSuite

```
55 fit_suite = FitSuite()
56 fit_suite.addSimulationAndRealData(simulation, real_data)
57 fit_suite.initPrint(10)
```

Line 55 creates a `FitSuite` object which provides the main interface to the minimization kernel of `BornAgain`. Line 56 submits simulation description and real data pair to the subsequent fitting. Line 57 sets up `FitSuite` to print on the screen the information about fit progress once per 10 iterations.

```

60     fit_suite.addFitParameter("*FormFactorCylinder/height", 4.*
        nanometer, 0.01*nanometer, AttLimits.lowerLimited(0.01))
61     fit_suite.addFitParameter("*FormFactorCylinder/radius", 6.*
        nanometer, 0.01*nanometer, AttLimits.lowerLimited(0.01))
62     fit_suite.addFitParameter("*FormFactorPrism3/height", 4.*
        nanometer, 0.01*nanometer, AttLimits.lowerLimited(0.01))
63     fit_suite.addFitParameter("*FormFactorPrism3/length", 12.*
        nanometer, 0.02*nanometer, AttLimits.lowerLimited(0.01))

```

Lines 60–63 enter the list of fitting parameters. Here we use the cylinders' height and radius and the prisms' height and side length. The cylinder's length and prism half side are initially equal to 4 nm, whereas the cylinder's radius and the prism half side length are equal to 6 nm before the minimization. The iteration step is equal to 0.01 nm and only the lower boundary is imposed to be equal to 0.01 nm.

Running the fit and accessing results

```

66     fit_suite.runFit()
67
68     print "Fitting completed."
69     fit_suite.printResults()
70     print "chi2:", fit_suite.getMinimizer().getMinValue()
71     fitpars = fit_suite.getFitParameters()
72     for i in range(0, fitpars.size()):
73         print fitpars[i].getName(), fitpars[i].getValue(),
            fitpars[i].getError()

```

Line 66 shows the command to start the fitting process. During the fitting the progress will be displayed on the screen. Lines 69–73 shows different ways of accessing the fit results.

More details about fitting, access to its results and visualization of the fit progress using `matplotlib` libraries can be learned from the following detailed example

```

./Examples/python/fitting/ex002_FitCylindersAndPrisms/
FitCylindersAndPrisms_detailed.py

```

4.3 Advanced fitting

4.3.1 Affecting χ^2 calculations

4.3.2 Simultaneous fits of several data sets

4.3.3 Using fitting strategies

4.3.4 Masking the real data

4.3.5 Tuning fitting algorithms

4.3.6 Fitting with correlated sample parameters

4.4 How to get the right answer from fitting

One of the main difficulties in fitting the data with the model is the presence of multiple local minima in the objective function. Many problems can cause the fit to fail, for example:

- an unreliable physical model,
- an inappropriate choice of objective function
- multiple local minima,
- an unphysical behavior of the objective function, unphysical regions in the parameters space,
- an unreliable parameter error calculation in the presence of limits on the parameter value,
- an exponential behavior of the objective function and the corresponding numerical inaccuracies, excessive numerical roundoff in the calculation of its value and derivatives,
- large correlations between parameters,
- very different scales of parameters involved in the calculation,
- not positive definite error matrix even at minimum.

The given list, of course, is not only related to BornAgain fitting. It remains applicable to any fitting program and any kind of theoretical model. Below we give some recommendations which might help the user to achieve reliable fit results.

General recommendations

- initially choose a small number of free fitting parameters,
- eliminate redundant parameters,

- provide a good initial guess for the fit parameters,
- start from the default minimizer settings and perform some fine tuning after some experience has been acquired,
- repeat the fit using different starting values for the parameters or their limits,
- repeat the fit, fixing and varying different groups of parameters,
to be continued...

Chapter 5

Software architecture

BornAgain is written in C++ and uses an object oriented approach to achieve modularity, extensibility and transparency. This leads to the task driven rather than the command driven approach in different aspects of the simulation and fitting of GISAS data. The user defines the sample structure, beam and detector characteristics and fit parameters using building blocks – classes – defined in core libraries of the framework. These buildings blocks are combined by the user according to his current task using one the following approaches:

- The user creates a Python script with a sample description and simulation settings using the BornAgain API. The user then runs the simulation by executing the script in the Python interpreter and assesses the simulation results using his preferred graphics or analysis library, e.g. Python + numpy + matplotlib.
- The user may write a standalone C++ application linked to the BornAgain libraries.
- The user interacts with the framework through a graphical user interface (forthcoming).

The object oriented approach in the software design allows users to have a much higher level of flexibility in the sample construction; it also decouples the building blocks used in the internal calculations and thereby facilitates the creation of new models, with little or no modification to the existing code.

The general structure of BornAgain and the way the user interacts with it are shown in Fig. 5.1. The framework consists of two shared libraries, `libBornAgainCore` and `libBornAgainFit`. Thanks to the Python interface they can be imported into Python as external modules. The library `libBornAgainCore` contains a number of classes, grouped into several class categories, necessary for the description of a model and running a simulation. The library `libBornAgainFit` contains a number of minimization engines and interfaces to them, allowing the user to fit real data with the model previously defined.

BornAgain depends on a few external and well established open-source libraries: boost, GNU scientific library, Eigen and Fast Fourier Transformation libraries. They are required to be installed on the system to run BornAgain on Unix Platforms. In the case of Windows

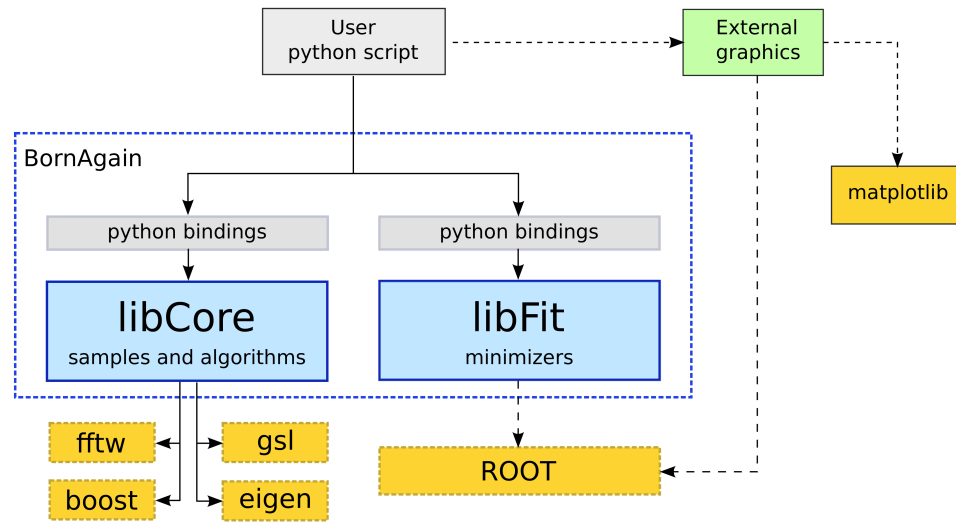


Figure 5.1: Structure of BornAgain libraries.

Platform they are added to the system automatically during BornAgain installation. Other libraries shown on the plot (ROOT, matplotlib) are optional.

5.1 Data classes for simulations and fits

This section will give an overview of the classes that are used to describe all the data needed to perform a single simulation. The prime elements of this data are formed by the sample, the experimental conditions (beam and detector parameters) and simulation parameters.

These classes constitute the main interface to the software's users, since they will mostly be interacting with the program by creating samples and running simulations with specific parameters. Since it is not the intent to explain internals of classes in this document, the text and figures will only mention the most important methods and fields of the classes discussed. Furthermore, getters and setters of private member fields will not be indicated, although these do belong to the public interface. For more detailed information about the project's classes, their methods and fields, the reader is referred to the source code documentation. REF?

5.1.1 The Experiment object

The `Experiment` class holds all references to data objects that are needed to perform a simulation. These consist in a sample description, possibly implemented by a builder object, detector and beam parameters and finally, a simulation parameter class that defines the different approximations that can be used during a simulation. Besides getters and setters for these fields, the class also contains a `runSimulation()` method that will generate

an ISimulation object that will perform the actual computations. The class diagram for Experiment is shown in figure 5.2.

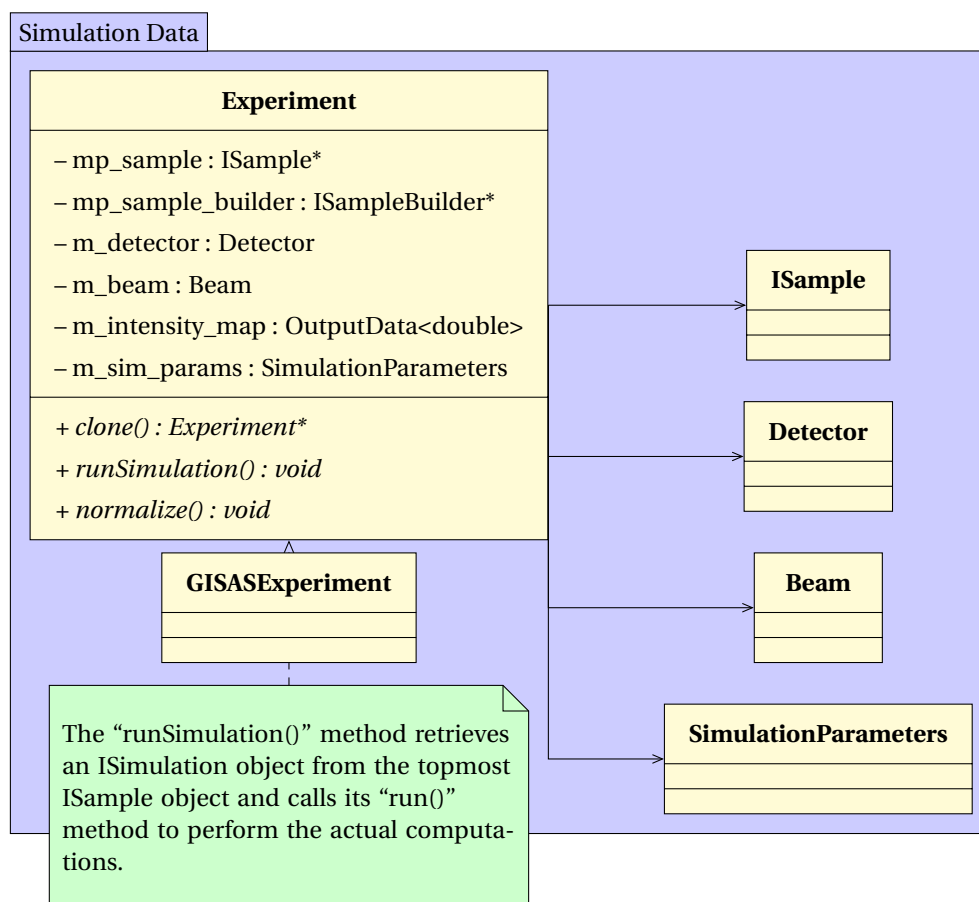


Figure 5.2: The Experiment class as a container for sample, beam, detector and simulation parameters.

5.1.2 The ISample class hierarchy

Samples are described by a hierarchical tree of objects which all adhere to the ISample interface. The composite pattern is used to achieve a common interface for all objects in the sample tree. The sample description is maximally decoupled from all computational classes, with the exception of the "createDWBASimulation()" method. This method will create a new object of type "DWBASimulation" that is capable of calculating the scattering contributions originating from the sample part in question. This coupling is not very tight however, since the ISample subclasses only need to know about which class to instantiate and return.

This interface and two of its subclasses are sketched in figure 5.3.

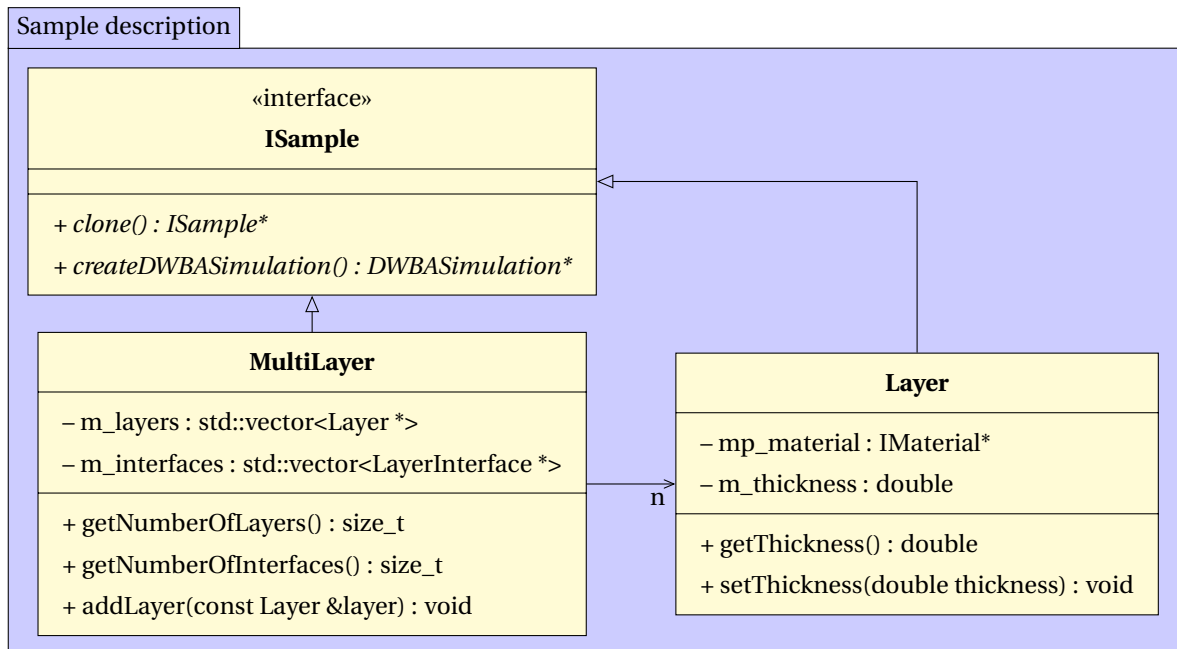


Figure 5.3: The ISample interface

5.1.3 The FitSuite class

5.1.4 The IMinimizer class

5.1.5 The MinimizerOptions class

Appendix A

Listings

A.1 Python simulation example

The following script can be found at

```
./Examples/python/simulation/ex001_CylindersAndPrisms/  
CylindersAndPrisms.py
```

```
1 import numpy
2 import matplotlib
3 import pylab
4 from libBornAgainCore import *
5
6
7 def get_sample():
8     """
9     Build and return the sample representing cylinders and
10     pyramids on top of
11     substrate without interference.
12     """
13     # defining materials
14     m_air = MaterialManager.getHomogeneousMaterial("Air", 0.0,
15     0.0)
16     m_substrate = MaterialManager.getHomogeneousMaterial("
17     Substrate", 6e-6, 2e-8)
18     m_particle = MaterialManager.getHomogeneousMaterial("Particle
19     ", 6e-4, 2e-8)
20
21     # collection of particles
22     cylinder_ff = FormFactorCylinder(5*nanometer, 5*nanometer)
23     cylinder = Particle(m_particle, cylinder_ff)
24     prism_ff = FormFactorPrism3(10*nanometer, 5*nanometer)
25     prism = Particle(m_particle, prism_ff)
26     particle_decoration = ParticleDecoration()
27     particle_decoration.addParticle(cylinder, 0.0, 0.5)
```

```
24     particle_decoration.addParticle(prism, 0.0, 0.5)
25     interference = InterferenceFunctionNone()
26     particle_decoration.addInterferenceFunction(interference)
27
28     # air layer with particles and substrate form multi layer
29     air_layer = Layer(m_air)
30     air_layer.setDecoration(particle_decoration)
31     substrate_layer = Layer(m_substrate, 0)
32     multi_layer = MultiLayer()
33     multi_layer.addLayer(air_layer)
34     multi_layer.addLayer(substrate_layer)
35     return multi_layer
36
37
38 def get_simulation():
39     """
40     Create and return GISAXS simulation with beam and detector
41     defined
42     """
43     simulation = Simulation()
44     simulation.setDetectorParameters(100, -1.0*degree, 1.0*degree
45                                     , 100, 0.0*degree, 2.0*degree, True)
46     simulation.setBeamParameters(1.0*angstrom, 0.2*degree, 0.0*
47                                 degree)
48     return simulation
49
50 def run_simulation():
51     """
52     Run simulation and plot results
53     """
54     sample = get_sample()
55     simulation = get_simulation()
56     simulation.setSample(sample)
57     simulation.runSimulation()
58     result = simulation.getIntensityData().getArray() + 1 # for
59                 log scale
60     pylab.imshow(numpy.rot90(result, 1), norm=matplotlib.colors.
61                 LogNorm(), extent=[-1.0, 1.0, 0, 2.0])
62     pylab.show()
63
64 if __name__ == '__main__':
65     run_simulation()
```

A.2 Python fitting example

The following script can be found at

```
./Examples/python/fitting/ex002_FitCylindersAndPrisms/
FitCylindersAndPrisms.py
```

```

1  from libBornAgainCore import *
2  from libBornAgainFit import *
3
4
5  def get_sample():
6      """
7      Build the sample representing cylinders and pyramids on top
8      of substrate without interference.
9      """
10     # defining materials
11     m_air = MaterialManager.getHomogeneousMaterial("Air", 0.0,
12     0.0)
13     m_substrate = MaterialManager.getHomogeneousMaterial("
14     Substrate", 6e-6, 2e-8)
15     m_particle = MaterialManager.getHomogeneousMaterial("Particle
16     ", 6e-4, 2e-8)
17
18     # collection of particles
19     cylinder_ff = FormFactorCylinder(1.0*nanometer, 1.0*nanometer
20     )
21     cylinder = Particle(m_particle, cylinder_ff)
22     prism_ff = FormFactorPrism3(2.0*nanometer, 1.0*nanometer)
23     prism = Particle(m_particle, prism_ff)
24     particle_decoration = ParticleDecoration()
25     particle_decoration.addParticle(cylinder, 0.0, 0.5)
26     particle_decoration.addParticle(prism, 0.0, 0.5)
27     interference = InterferenceFunctionNone()
28     particle_decoration.addInterferenceFunction(interference)
29
30     # air layer with particles and substrate form multi layer
31     air_layer = Layer(m_air)
32     air_layer.setDecoration(particle_decoration)
33     substrate_layer = Layer(m_substrate, 0)
34     multi_layer = MultiLayer()
35     multi_layer.addLayer(air_layer)
36     multi_layer.addLayer(substrate_layer)
37     return multi_layer
38
39
40 def get_simulation():
41     """
42     Create GISAXS simulation with beam and detector defined
43     """

```

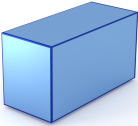





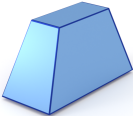


```
39     simulation = Simulation()
40     simulation.setDetectorParameters(100, -1.0*degree, 1.0*degree
41         , 100, 0.0*degree, 2.0*degree, True)
42     simulation.setBeamParameters(1.0*angstrom, 0.2*degree, 0.0*
43         degree)
44     return simulation
45
46 def run_fitting():
47     """
48     run fitting
49     """
50     sample = get_sample()
51     simulation = get_simulation()
52     simulation.setSample(sample)
53
54     real_data = OutputDataIOFactory.readIntensityData('
55         refdata_fitcylinderprisms.txt')
56
57     fit_suite = FitSuite()
58     fit_suite.addSimulationAndRealData(simulation, real_data)
59     fit_suite.initPrint(10)
60
61     # setting fitting parameters with starting values
62     fit_suite.addFitParameter("*FormFactorCylinder/height", 4.*
63         nanometer, 0.01*nanometer, AttLimits.lowerLimited(0.01))
64     fit_suite.addFitParameter("*FormFactorCylinder/radius", 6.*
65         nanometer, 0.01*nanometer, AttLimits.lowerLimited(0.01))
66     fit_suite.addFitParameter("*FormFactorPrism3/height", 4.*
67         nanometer, 0.01*nanometer, AttLimits.lowerLimited(0.01))
68     fit_suite.addFitParameter("*FormFactorPrism3/length", 12.*
69         nanometer, 0.02*nanometer, AttLimits.lowerLimited(0.01))
70
71     # running fit
72     fit_suite.runFit()
73
74     print "Fitting completed."
75     fit_suite.printResults()
76     print "chi2:", fit_suite.getMinimizer().getMinValue()
77     fitpars = fit_suite.getFitParameters()
78     for i in range(0, fitpars.size()):
79         print fitpars[i].getName(), fitpars[i].getValue(),
80             fitpars[i].getError()
81
82 if __name__ == '__main__':
83     run_fitting()
```

Appendix B




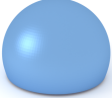


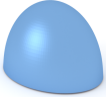
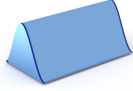

Form factors

Table B.1 lists the particles shapes whose form factors have been implemented in BornAgain.

Table B.1: Table of form factors implemented in BornAgain.

Shape	Shape	Shape
Box, Section B.1	Prism3, Section B.2	Tetrahedron, Section B.3
		
Prism6, Section B.4	Cone6, Section B.5	Pyramid, Section B.6
		
Anisotropic pyramid, Section B.7	Cuboctahedron, Section B.8	Cylinder, Section B.9
		

Appendix B. Form factors

Shape	Shape	Shape
Ellipsoidal cylinder, Section B.10	Cone, Section B.11	Full Sphere, Section B.12
		
Truncated Sphere, Section B.13	Full Spheroid, Section B.14	Truncated Spheroid, Section B.15
		
Hemi Ellipsoid, Section B.16	Ripple1, Section B.17	Ripple2, Section B.18
		

In BornAgain the form factor is defined as

$$F(\mathbf{q}) = \int_V \exp(i\mathbf{q} \cdot \mathbf{r}) d^3\mathbf{r}, \quad (\text{B.1})$$

where V is the volume of the particle's shape, $\mathbf{q} = \mathbf{k}_i - \mathbf{k}_f$ is the scattering vector with \mathbf{k}_f and \mathbf{k}_i the scattered and incident wave vector, respectively.

The particle's shape is parametrized in a cartesian frame, with its z -axis pointing upwards and its origin at the center of the bottom of the particle: $\mathbf{r} = (x, y, z)$. In the followings, a schematic view will depict this layout for each form factor.

All form factors have been implemented with complex scattering vectors in order to take any material absorption into account.

The particles can be rotated in a different direction by using one of the following transformations: `CreateRotateX(θ)`, `CreateRotateY(θ)`, `CreateRotateZ(θ)`, where capital X, Y, Z mark rotations around the associated axis and θ is the angle of rotation from this axis. For example, in order to rotate a pyramid by 45° around z -axis, the user could use the following Python script:

Appendix B. Form factors

```
pyramid_ff = FormFactorPyramid(10*nanometer, 5*nanometer,
    deg2rad(54.73 ) )
pyramid = Particle(m_particle, pyramid_ff)
angle_around_z = 45.*degree
transform = Transform3D.createRotateZ(angle_around_z)
particle_decoration = ParticleDecoration()
particle_decoration.addParticle(pyramid, transform)
```

B.1 Box

B.1.1 Real-space geometry

This shape is a rectangular cuboid as shown in fig. B.1.

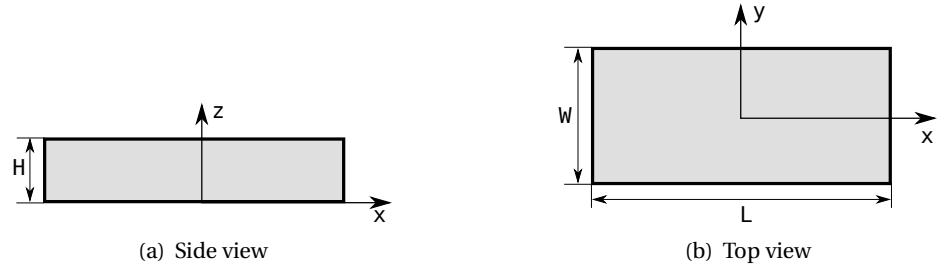


Figure B.1: Sketch of a Box.

Parameters:

- length of the base L ,
- width of the base W ,
- height H .

Properties:

- volume $V = LWH$,
- particle surface seen from above $S = LW$.

B.1.2 Expression of the form factor

$$F_{\text{Box}}(\mathbf{q}, L, W, H) = LWH \exp\left(i q_z \frac{H}{2}\right) \text{sinc}\left(q_x \frac{L}{2}\right) \text{sinc}\left(q_y \frac{W}{2}\right) \text{sinc}\left(q_z \frac{H}{2}\right),$$

where $\text{sinc}(x) = \sin(x)/x$ is the cardinal sine.

Syntax: `FormFactorBox(length, width, height)`

B.1.3 Examples

Figure B.2 shows the normalized intensity $|F|^2/V^2$, computed with $L = 20$ nm, $W = 16$ nm, $H = 13$ nm, and $\alpha = 60^\circ$:

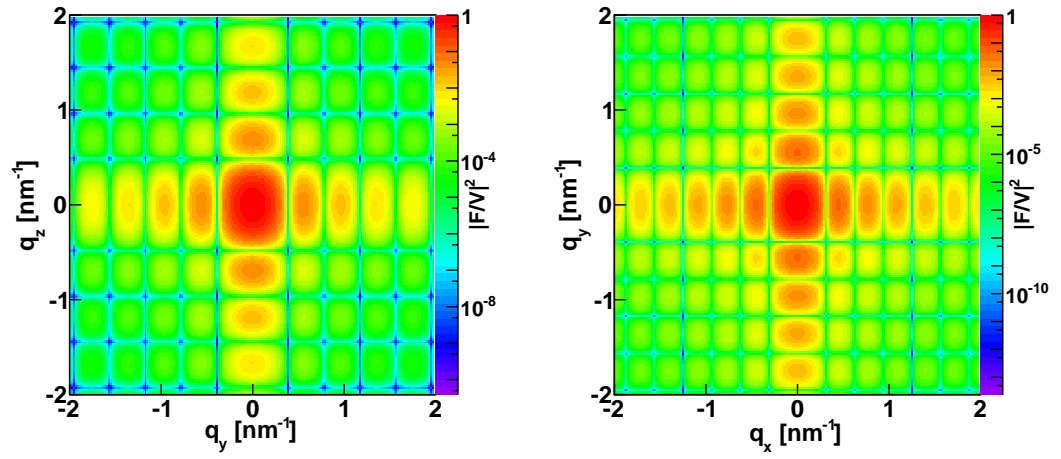


Figure B.2: Normalized intensity for the form factor of a Box $|F|^2/V^2$, plotted against (q_z, q_y) and (q_x, q_y) and computed with $L = 20$ nm, $W = 16$ nm, and $H = 13$ nm.

B.2 Prism3

B.2.1 Real-space geometry

This shape is a triangular prism, whose base is an equilateral triangle as shown in fig. B.3.

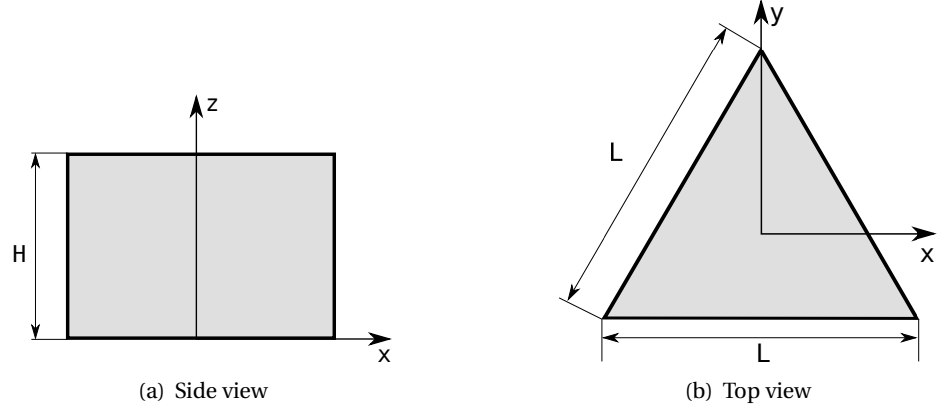


Figure B.3: Sketch of a Prism3.

Parameters:

- length L of one side of the base,
- height H .

Properties:

- volume $V = \frac{\sqrt{3}}{4}HL^2$,
- particle surface seen from above $S = \frac{\sqrt{3}}{4}L^2$.

B.2.2 Expression of the form factor

$$F_{\text{Prism3}}(\mathbf{q}, L, H) = \frac{2\sqrt{3}}{q_x^2 - 3q_y^2} \exp\left(-iq_y \frac{L}{2\sqrt{3}}\right) \left[\exp\left(i\sqrt{3}q_y \frac{L}{2}\right) - \cos\left(q_x \frac{L}{2}\right) - i\sqrt{3}q_y \frac{L}{2} \text{sinc}\left(q_x \frac{L}{2}\right) \right] \\ \times H \text{sinc}\left(q_z \frac{H}{2}\right) \exp\left(iq_z \frac{H}{2}\right),$$

where $\text{sinc}(x) = \sin(x)/x$ is the cardinal sine.

Syntax: FormFactorPrism3(length, height)

B.2.3 Examples

Figure B.4 shows the normalized intensity $|F|^2/V^2$, computed with $L = 10$ nm and $H = 13$ nm.

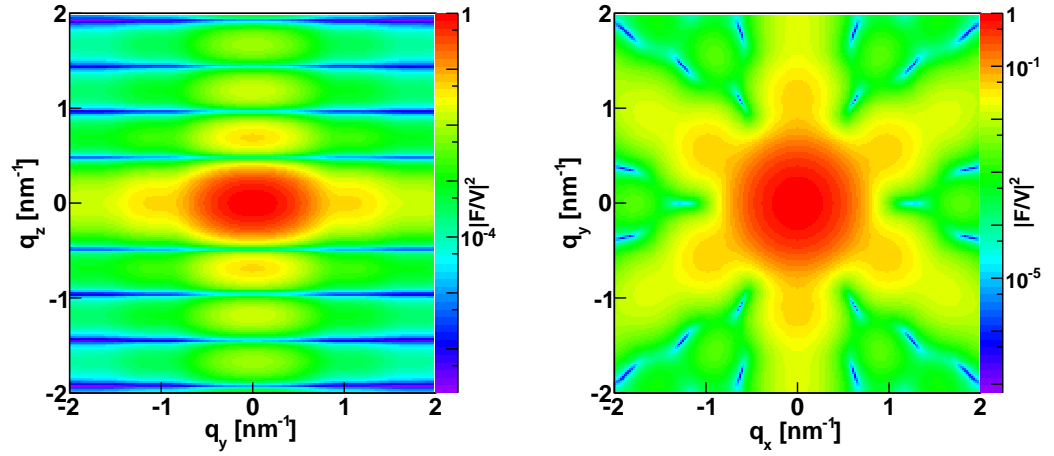


Figure B.4: Normalized intensity for the form factor of a Prism3 $|F|^2/V^2$, plotted against (q_z, q_y) and (q_x, q_y) and computed with $L = 10$ nm and $H = 13$ nm.

B.3 Tetrahedron

B.3.1 Real-space geometry

This shape is a truncated tetrahedron as shown in fig. B.5.

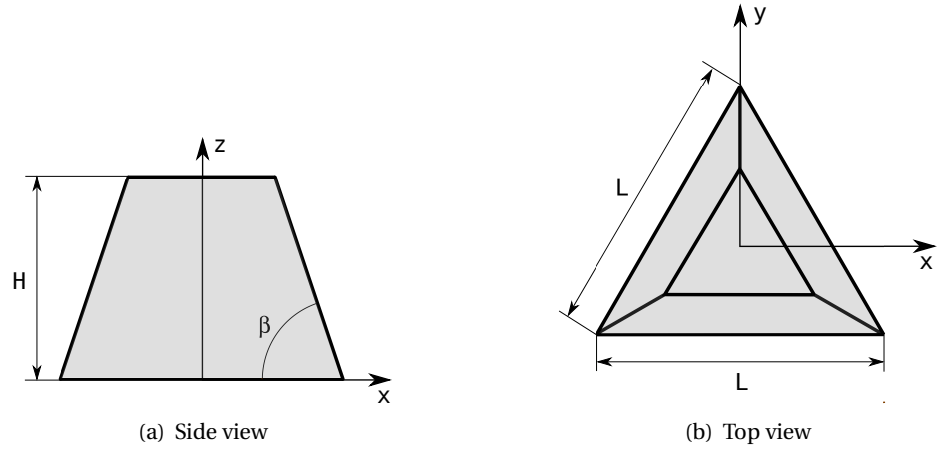


Figure B.5: Sketch of a Tetrahedron. The implementation of this shape uses angle α , which is linked to β via $\tan \alpha = 2 \tan \beta$. α is measured along one of the base lines and β at one of the base vertices.

Parameters:

- length of one side of the equilateral triangular base L ,
- height H ,
- angle α is the angle between the base and the side faces, taken in the middle of the base lines.

Restrictions on the parameters: $\frac{H}{L} < \frac{\tan \alpha}{2\sqrt{3}}$.

Properties:

- volume $V = \frac{\tan(\alpha)}{24} L^3 \left[1 - \left(1 - \sqrt{3} \frac{2H}{L \tan(\alpha)} \right)^3 \right]$,
- particle surface seen from above $S = \frac{\sqrt{3}}{4} L^2$.

B.3.2 Expression of the form factor

$$F_{\text{Tetrahedron}}(\mathbf{q}, L, H, \alpha) = \frac{\sqrt{3}H}{q_x(q_x^2 - 3q_y^2)} \exp\left(iq_z \frac{L}{2 \tan(\alpha) \sqrt{3}}\right) \times \\ \left\{ 2q_x \exp(iq_3 D) \text{sinc}(q_3 H) - (q_x + \sqrt{3}q_y) \exp(iq_1 D) \text{sinc}(q_1 D) - (q_x - \sqrt{3}q_y) \exp(-iq_2 D) \text{sinc}(q_2 H) \right\},$$

with $\text{sinc}(x) = \sin(x)/x$,

$$q_1 = \frac{1}{2} \left[\frac{q_x \sqrt{3} - q_y}{\tan \alpha} - q_z \right], \quad q_2 = \frac{1}{2} \left[\frac{q_x \sqrt{3} + q_y}{\tan \alpha} + q_z \right], \quad q_3 = \frac{q_y}{\tan \alpha} - \frac{q_z}{2}, \quad D = \frac{L \tan \alpha}{\sqrt{3}} - H.$$

Syntax: FormFactorTetrahedron(length, height, alpha)

B.3.3 Examples

Figure B.6 shows the normalized intensity $|F|^2/V^2$, computed with $L = 15$ nm, $H = 6$ nm and $\alpha = 60^\circ$.

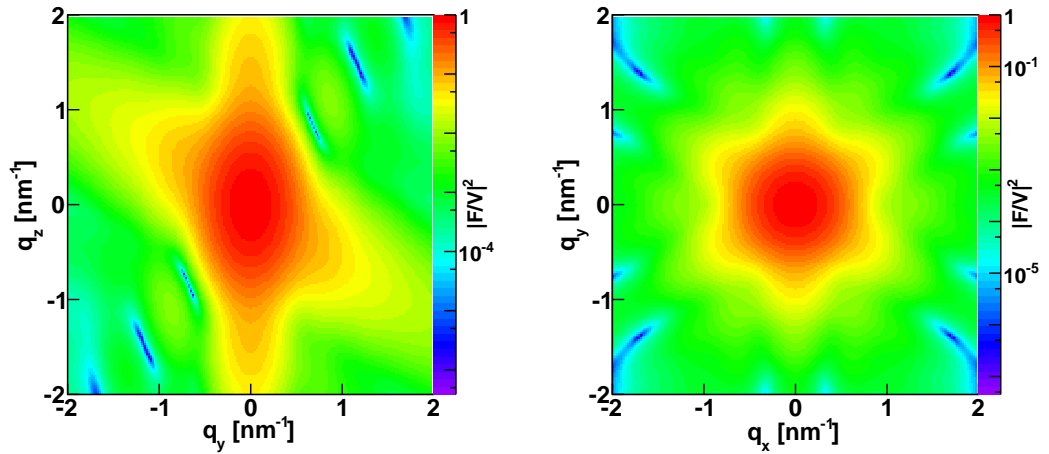


Figure B.6: Normalized intensity for the form factor of a Tetrahedron $|F|^2/V^2$, plotted against (q_z, q_y) and (q_x, q_y) and computed with $L = 15$ nm, $H = 6$ nm and $\alpha = 60^\circ$.

B.4 Prism6

B.4.1 Real-space geometry

This shape is an hexagonal prism (see fig. B.7).

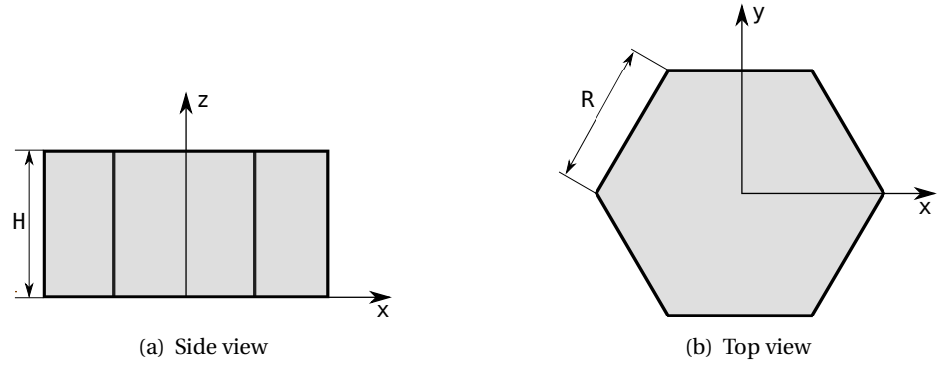


Figure B.7: Sketch of a Prism6.

Parameters:

- radius of the hexagonal base R ,
- height H .

Properties:

- volume $V = \frac{3\sqrt{3}}{2}HR^2$,
- particle surface seen from above $S = \frac{3\sqrt{3}R^2}{2}$.

B.4.2 Expression of the form factor

$$F_{\text{Prism6}}(\mathbf{q}, R, H) = \frac{4H\sqrt{3}}{3q_y^2 - q_x^2} \text{sinc}\left(q_z \frac{H}{2}\right) \exp\left(-iq_z \frac{H}{2}\right) \times \\ \left\{ \frac{3q_y^2 R^2}{4} \text{sinc}\left(\frac{q_x R}{2}\right) \text{sinc}\left(\frac{\sqrt{3}q_y R}{2}\right) + \cos(q_x R) - \cos\left(q_y \frac{\sqrt{3}R}{2}\right) \cos\left(\frac{q_x R}{2}\right) \right\},$$

with $\text{sinc}(x) = \sin(x)/x$.

Syntax: `FormFactorPrism6(radius, height)`

B.4.3 Examples

Figure B.8 shows the normalized intensity $|F|^2/V^2$, computed with $R = 5$ nm and $H = 11$ nm.

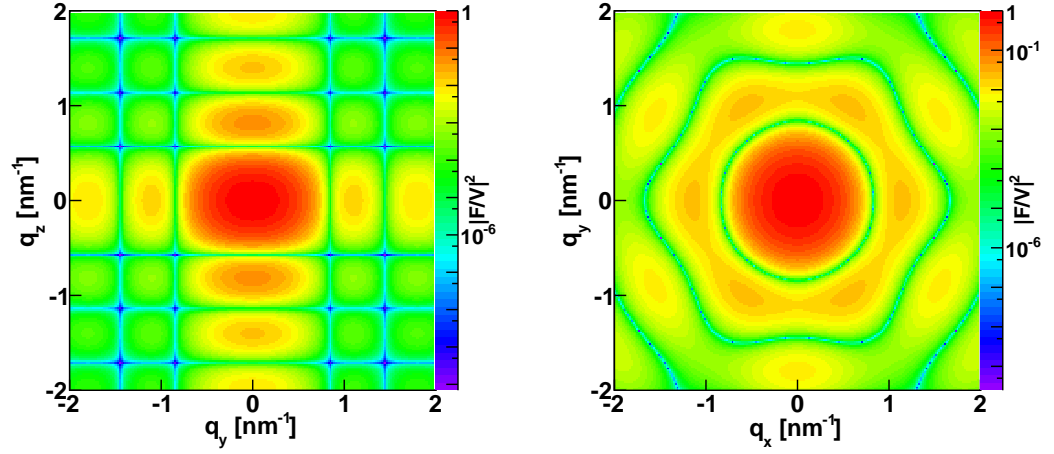


Figure B.8: Normalized intensity for the form factor of a Prism6 $|F|^2/V^2$, plotted against (q_z, q_y) and (q_x, q_y) and computed with $R = 5$ nm and $H = 11$ nm.

B.5 Cone6

B.5.1 Real-space geometry

It is a truncated hexagonal pyramid (see fig. B.9).

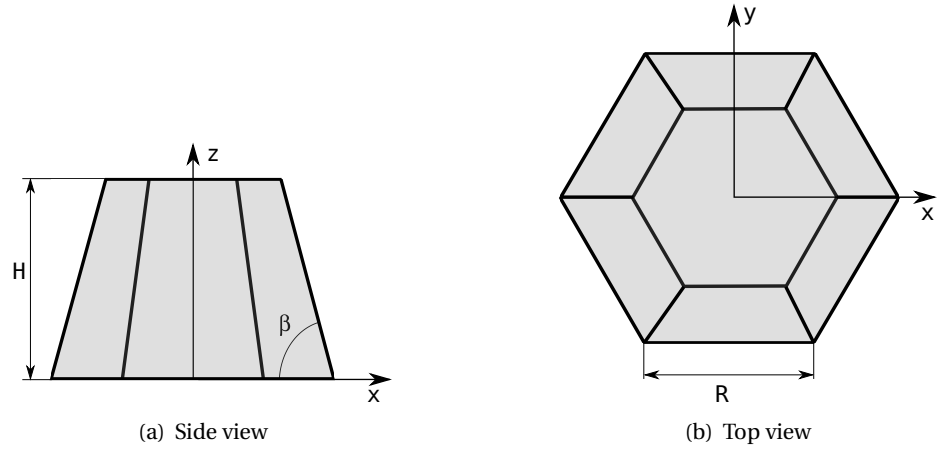


Figure B.9: Sketch of a Cone6. The implementation of this shape uses angle α , which is linked to β via $\tan \alpha = \frac{2}{\sqrt{3}} \tan \beta$. α is measured along one of the base lines and β at one of the base vertices.

Parameters:

- radius of the regular hexagonal base R ,
- height H ,
- angle α is considered between one of the side faces and the middle of a base length.

Restrictions on the parameters: $\frac{2H}{\sqrt{3}R} < \tan \alpha$.

Properties:

- volume $V = \frac{3}{4} \tan(\alpha) R^3 \left[1 - \left(1 - \frac{2H}{\tan(\alpha) R \sqrt{3}} \right)^3 \right]$,
- particle surface seen from above $S = \frac{3\sqrt{3}R^2}{2}$.

B.5.2 Expression of the form factor

The calculation can be derived from “Prism6” (Section B.4) by considering a side length varying with the vertical position:

$$F_{\text{Cone6}}(\mathbf{q}, R, H, \alpha) = \frac{4\sqrt{3}}{3q_y^2 - q_x^2} \int_0^H \exp(iq_z z) \left[\frac{3}{4} R_z^2 q_y^2 \operatorname{sinc}\left(\frac{q_x R_z}{2}\right) \operatorname{sinc}\left(\frac{\sqrt{3} q_y R_z}{2}\right) + \cos(q_x R_z) - \cos\left(\frac{\sqrt{3} q_y R_z}{2}\right) \cos\left(\frac{q_x R_z}{2}\right) \right] dz$$

with $R_z = R - \frac{2z}{\sqrt{3}\tan(\alpha)}$ and $\operatorname{sinc}(x) = \sin(x)/x$.

Syntax: FormFactorCone6(radius,height, alpha)

B.5.3 Examples

Figure B.10 shows the normalized intensity $|F|^2/V^2$, computed with $R = 10$ nm, $H = 13$ nm, and $\alpha = 60^\circ$.

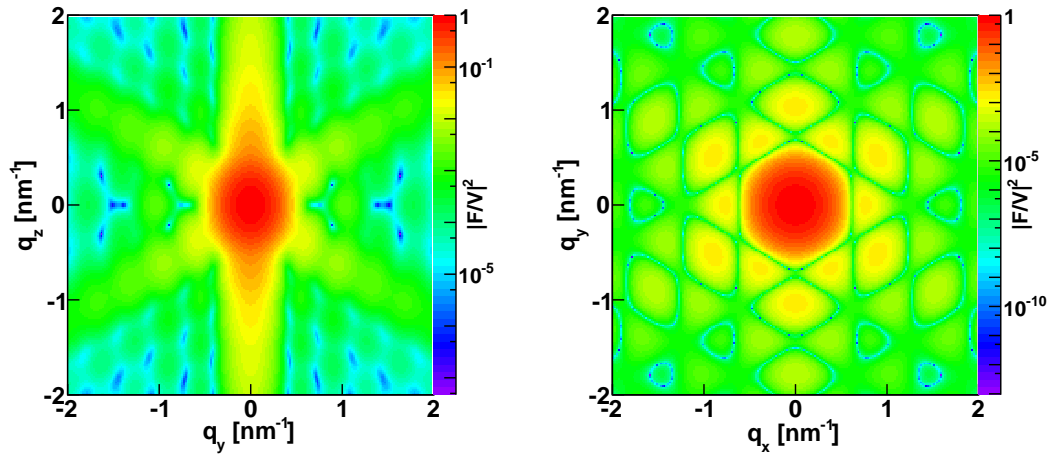


Figure B.10: Normalized intensity for the form factor of a Cone6 $|F|^2/V^2$, plotted against (q_z, q_y) and (q_x, q_y) and computed with $R = 10$ nm, $H = 13$ nm, and $\alpha = 60^\circ$.

B.6 Pyramid

B.6.1 Real-space geometry

This shape is a truncated pyramid with a square base as shown in fig. B.11.

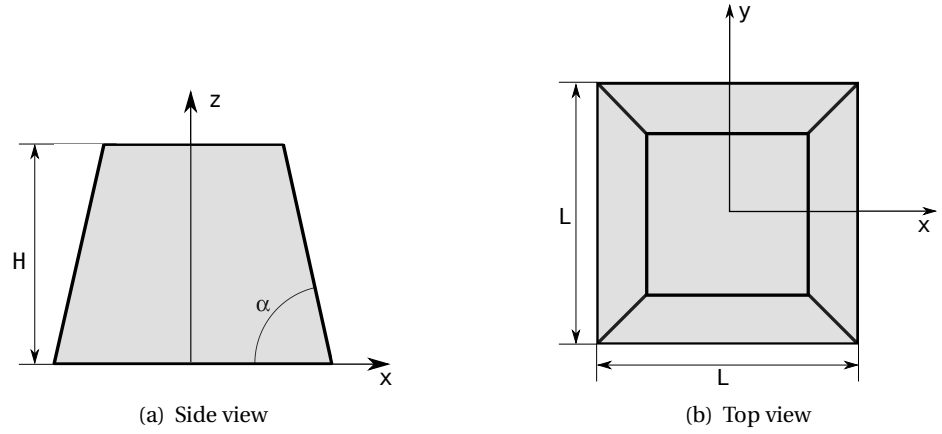


Figure B.11: Sketch of a Pyramid

Parameters:

- length of one side of the square base L ,
- height H ,
- α is the angle between the base and the side faces, taken in the middle of the base lines.

Restrictions on the parameters: $\frac{2H}{L} < \tan(\alpha)$.

Properties:

- volume $V = \frac{1}{6} \tan(\alpha) L^3 \left[1 - \left(1 - \frac{2H}{\tan(\alpha)L} \right)^3 \right]$,
- particle surface seen from above $S = L^2$.

B.6.2 Expression of the form factor

$$F_{\text{Pyramid}}(\mathbf{q}, L, H, \alpha) = \frac{H}{q_x q_y} \times \left\{ K_1 \cos \left[(q_x - q_y) \frac{L}{2} \right] + K_2 \sin \left[(q_x - q_y) \frac{L}{2} \right] - K_3 \cos \left[(q_x + q_y) \frac{L}{2} \right] - K_4 \sin \left[(q_x + q_y) \frac{L}{2} \right] \right\},$$

with $\text{sinc}(x) = \sin(x)/x$,

$$\begin{aligned} q_1 &= \frac{1}{2} \left[\frac{q_x - q_y}{\tan(\alpha)} + q_z \right], & q_2 &= \frac{1}{2} \left[\frac{q_x - q_y}{\tan(\alpha)} - q_z \right] \\ q_3 &= \frac{1}{2} \left[\frac{q_x + q_y}{\tan(\alpha)} + q_z \right], & q_4 &= \frac{1}{2} \left[\frac{q_x + q_y}{\tan(\alpha)} - q_z \right] \\ K_1 &= \text{sinc}(q_1 H) \exp(i q_1 H) + \text{sinc}(q_2 H) \exp(-i q_2 H) \\ K_2 &= -i \text{sinc}(q_1 H) \exp(i q_1 H) + i \text{sinc}(q_2 H) \exp(-i q_2 H) \\ K_3 &= \text{sinc}(q_3 H) \exp(i q_3 H) + \text{sinc}(q_4 H) \exp(-i q_4 H) \\ K_4 &= -i \text{sinc}(q_3 H) \exp(i q_3 H) + i \text{sinc}(q_4 H) \exp(-i q_4 H) \end{aligned}$$

Syntax: FormFactorPyramid(length, height, alpha)

B.6.3 Examples

Figure B.12 shows the normalized intensity $|F|^2/V^2$, computed with $L = 18$ nm, $H = 13$ nm and $\alpha = 60^\circ$.

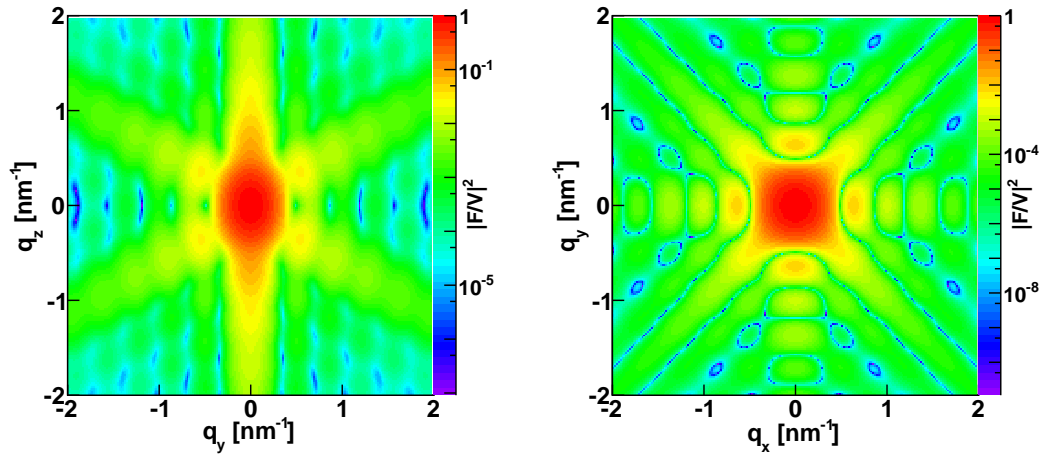


Figure B.12: Normalized intensity for the form factor of a pyramid $|F|^2/V^2$, plotted against (q_z, q_y) and (q_x, q_y) and computed with $L = 18$ nm and $H = 13$ nm, and $\alpha = 60^\circ$.

B.7 Anisotropic pyramid

B.7.1 Real-space geometry

This shape is a truncated right pyramid with a rectangular base as shown in fig. B.13.

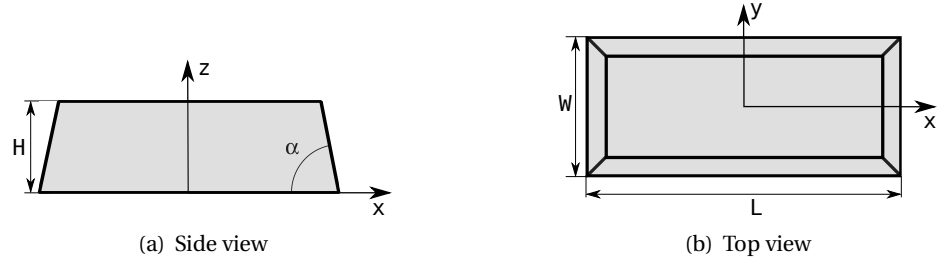


Figure B.13: Sketch of an Anisotropic Pyramid.

Parameters:

- full length of the base L ,
- full width of the base W ,
- height H ,
- α is the angle between the base and the side faces, taken in the middle of the base lines.

Restrictions on the parameters: $\frac{2H}{L} < \tan(\alpha)$ and $\frac{2H}{W} < \tan(\alpha)$.

Properties:

- volume $V = H \left[LW - \frac{(L+W)H}{\tan(\alpha)} + \frac{4}{3} \frac{H^2}{\tan^2(\alpha)} \right]$,
- particle surface seen from above $S = LW$.

B.7.2 Expression of the form factor

$$F_{\text{AnisoPyramid}}(\mathbf{q}, L, W, H, \alpha) = \frac{H}{q_x q_y} \times \left\{ K_1 \cos\left(q_x \frac{L}{2} - q_y \frac{W}{2}\right) + K_2 \sin\left(q_x \frac{L}{2} - q_y \frac{W}{2}\right) - K_3 \cos\left(q_x \frac{L}{2} + q_y \frac{W}{2}\right) - K_4 \sin\left(q_x \frac{L}{2} + q_y \frac{W}{2}\right) \right\},$$

with $\text{sinc}(x) = \sin(x)/x$,

$$\begin{aligned} K_1 &= \exp(-i q_2 H) \text{sinc}(q_2 H) + \exp(i q_1 H) \text{sinc}(q_1 H) \\ K_2 &= i \exp(-i q_2 H) \text{sinc}(q_2 H) - i \exp(i q_1 H) \text{sinc}(q_1 H) \\ K_3 &= \exp(-i q_4 H) \text{sinc}(q_4 H) + \exp(i q_3 H) \text{sinc}(q_3 H) \\ K_4 &= i \exp(i q_4 H) \text{sinc}(q_4 H) - i \exp(i q_3 H) \text{sinc}(q_3 H) \\ q_1 &= \frac{1}{2} \left[\frac{q_x - q_y}{\tan \alpha} + q_z \right], \quad q_2 = \frac{1}{2} \left[\frac{q_x - q_y}{\tan \alpha} - q_z \right] \\ q_3 &= \frac{1}{2} \left[\frac{q_x + q_y}{\tan \alpha} + q_z \right], \quad q_4 = \frac{1}{2} \left[\frac{q_x + q_y}{\tan \alpha} - q_z \right] \end{aligned}$$

Syntax: FormFactorAnisoPyramid(length, width, height, alpha)

B.7.3 Examples

Figure B.14 shows the normalized intensity $|F|^2/V^2$, computed with $L = 20$ nm, $W = 16$ nm, $H = 13$ nm, and $\alpha = 60^\circ$.

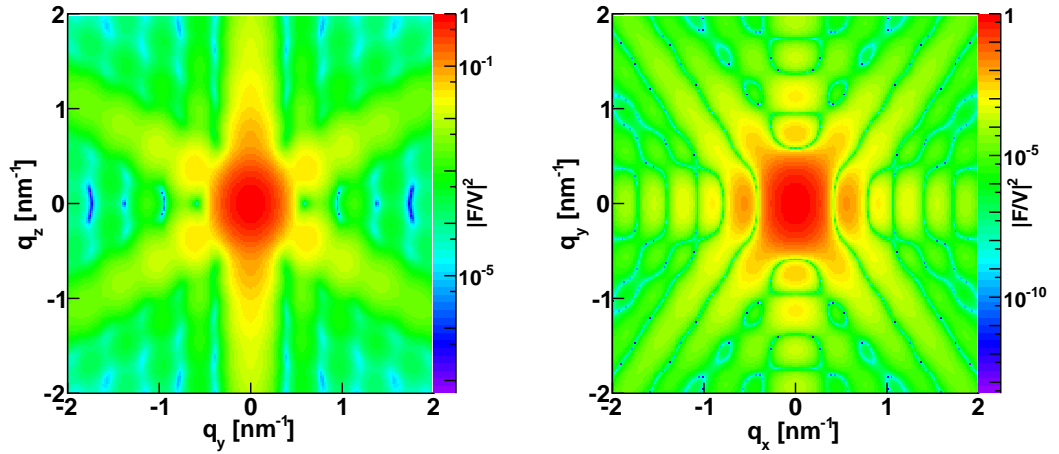


Figure B.14: Normalized intensity for the form factor of an anisotropic pyramid $|F|^2/V^2$, plotted against (q_z, q_y) and (q_x, q_y) and computed with $L = 20$ nm, $W = 16$ nm, $H = 13$ nm, and $\alpha = 60^\circ$.

B.8 Cuboctahedron

B.8.1 Real-space geometry

It is a combination of two pyramids with square bases, as shown in fig. B.15: the bottom one is upside down with an height H and the top one has the opposite orientation (the standard one) and an height $r_H H$.

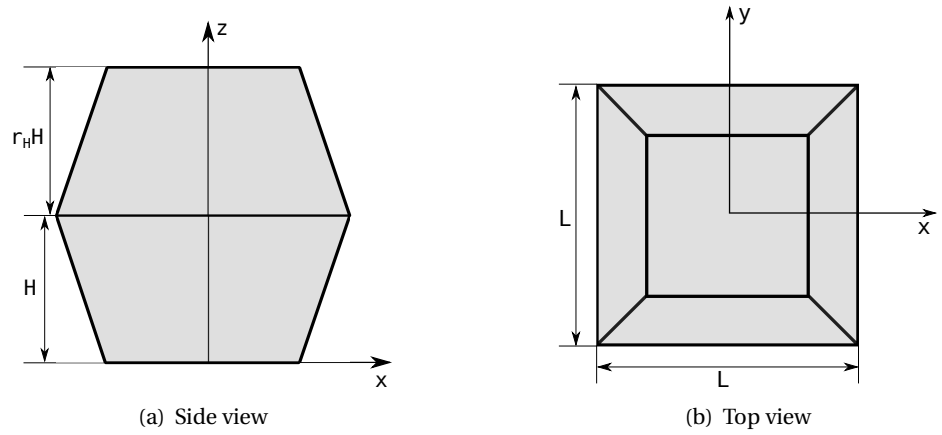


Figure B.15: Sketch of a Cuboctahedron.

Parameters:

- length of the shared square base L ,
- height H ,
- height_ratio r_H ,
- α is the angle between the base and the side faces, taken in the middle of the base lines (see fig. B.11 in Section B.6).

Restrictions on the parameters: $\frac{2H}{L} < \tan(\alpha)$ and $\frac{2r_H H}{L} < \tan(\alpha)$.

Properties:

- volume $V = \frac{1}{6} \tan(\alpha) L^3 \left[2 - \left(1 - \frac{2H}{L \tan(\alpha)} \right)^3 - \left(1 - \frac{2r_H H}{L \tan(\alpha)} \right)^3 \right]$,
- particle surface seen from above $S = L^2$.

B.8.2 Expression of the form factor

$$F_{\text{Cuboctahedron}}(\mathbf{q}, L, H, r_H, \alpha) = \exp(i q_z H) \left[F_{\text{Pyramid}}(q_x, q_y, q_z, L, r_H H, \alpha) + F_{\text{Pyramid}}(q_x, q_y, -q_z, L, H, \alpha) \right]$$

Syntax: FormFactorCuboctahedron(length, height, height_ratio, alpha)

B.8.3 Examples

Figure B.16 shows the normalized intensity $|F|^2/V^2$, computed with $L = 20$ nm, $H = 13$ nm, $r_H = 0.7$, and $\alpha = 60^\circ$.

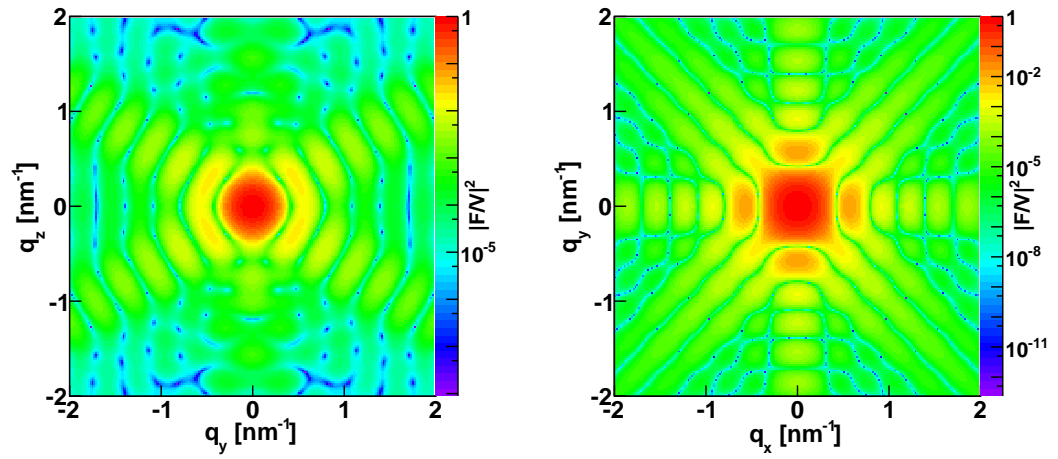


Figure B.16: Normalized intensity for the form factor of a cuboctahedron $|F|^2/V^2$, plotted against (q_z, q_y) and (q_x, q_y) and computed with $L = 20$ nm, $H = 13$ nm, $r_H = 0.7$, and $\alpha = 60^\circ$.

B.9 Cylinder

B.9.1 Real-space geometry

This shape is a right circular cylinder (see fig. B.17).

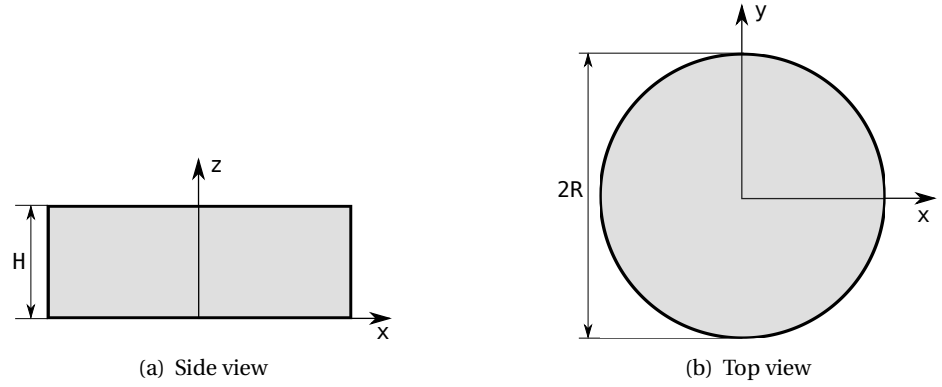


Figure B.17: Sketch of a Cylinder.

Parameters:

- radius of the circular base R .
- height H .

Properties:

- volume $V = \pi R^2 H$,
- particle surface seen from above $S = \pi R^2$.

B.9.2 Expression of the form factor

$$F_{\text{Cylinder}}(\mathbf{q}, R, H) = 2\pi R^2 H \operatorname{sinc}\left(q_z \frac{H}{2}\right) \exp\left(i q_z \frac{H}{2}\right) \frac{J_1(q_{\parallel} R)}{q_{\parallel} R},$$

with $q_{\parallel} = \sqrt{q_x^2 + q_y^2}$ and $J_1(x)$ is the first order Bessel function of the first kind [9].

Syntax: `FormFactorCylinder(radius, height)`

B.9.3 Examples

Figure B.18 shows the normalized intensity $|F|^2/V^2$, computed with $R = 8$ nm and $H = 16$ nm.

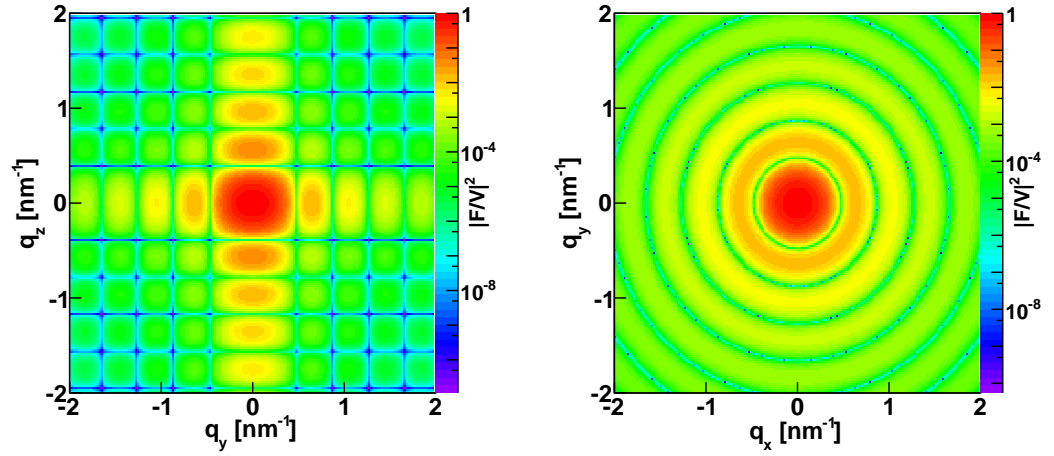


Figure B.18: Normalized intensity for the form factor of a cylinder $|F|^2/V^2$, plotted against (q_z, q_y) and (q_x, q_y) . It has been computed with $R = 8$ nm and $H = 16$ nm.

B.10 Ellipsoidal cylinder

B.10.1 Real-space geometry

This is a cylinder whose cross section is an ellipse.

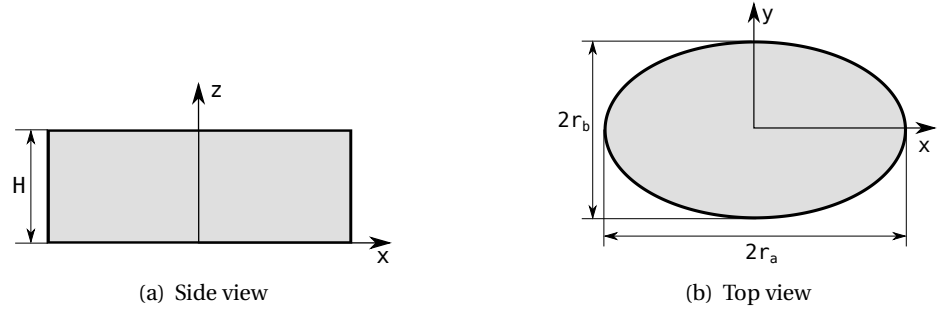


Figure B.19: Sketch of an Ellipsoidal Cylinder.

Parameters:

- r_a = half length of the ellipse main axis parallel to x ,
- r_b = half length of the ellipse main axis parallel to y ,
- height H .

Properties:

- volume $V = \pi r_a r_b H$,
- particle surface seen from above $S = r_a r_b$.

B.10.2 Expression of the form factor

The total form factor is given by

$$F_{\text{EllipsoidalCylinder}}(\mathbf{q}, R, W, H) = 2\pi r_a r_b H \exp\left(i \frac{q_z H}{2}\right) \text{sinc}\left(\frac{q_z H}{2}\right) \frac{J_1(\gamma)}{\gamma},$$

with $\gamma = \sqrt{(q_x r_a)^2 + (q_y r_b)^2}$ and $J_1(x)$ is the first order Bessel function of the first kind [9].

Syntax: `FormFactorEllipsoidalCylinder(r_a , r_b , height)`

B.10.3 Examples

Figure B.20 shows the normalized intensity $|F|^2/V^2$, computed with $r_a = 13$ nm, $r_b = 8$ nm, and $H = 16$ nm.

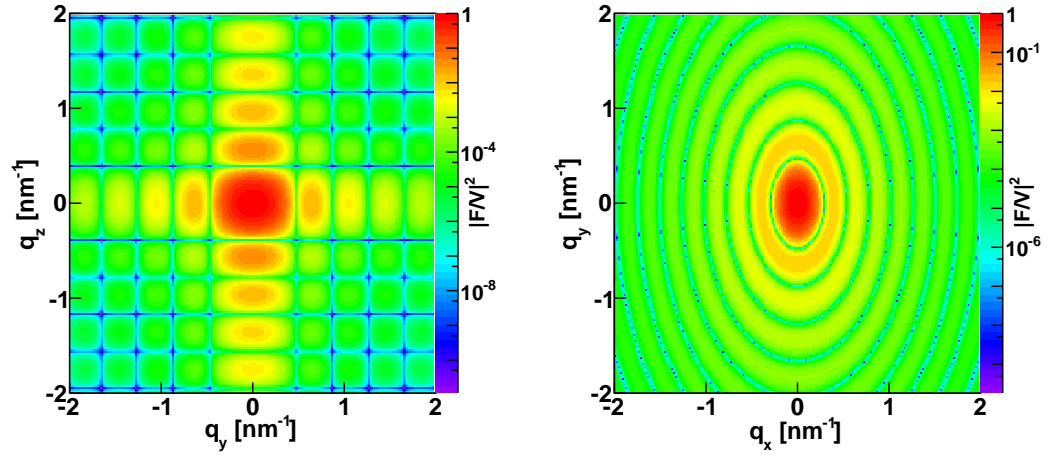


Figure B.20: Normalized intensity for the form factor of an ellipsoidal cylinder $|F|^2/V^2$, plotted against (q_z, q_y) and (q_x, q_y) and computed with $r_a = 8$ nm, $r_b = 13$ nm, and $H = 16$ nm.

B.11 Cone

B.11.1 Real-space geometry

This shape is a truncated cone as shown in fig. B.21.

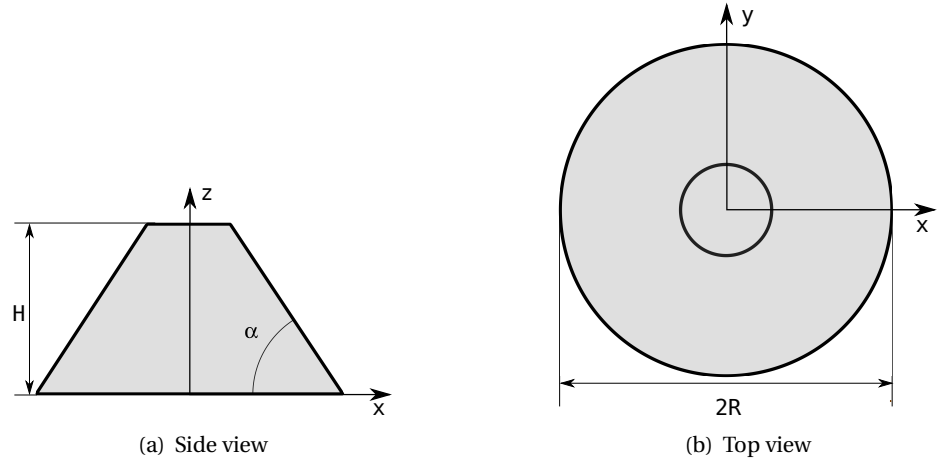


Figure B.21: Sketch of a Cone.

Parameters:

- radius R ,
- height H ,
- α is the angle between the side and the circular base.

Restrictions on the parameters: $\frac{H}{R} < \tan(\alpha)$.

Properties:

- volume $V = \frac{\pi}{3} \tan(\alpha) R^3 \left[1 - \left(1 - \frac{H}{\tan(\alpha) R} \right)^3 \right]$,
- particle surface seen from above $S = \pi R^2$.

B.11.2 Expression of the form factor

$$F_{\text{Cone}}(\mathbf{q}, R, H, \alpha) = \int_0^H 2\pi R_z^2 \frac{J_1(q_{\parallel} R_z)}{q_{\parallel} R_z} \exp(i q_z z) dz,$$

with $R_z = R - \frac{z}{\tan \alpha}$, $\mathbf{q}_{\parallel} = \sqrt{q_x^2 + q_y^2}$ and $J_1(x)$ is the first order Bessel function of the first kind [9].

Syntax: FormFactorCone(radius, height, alpha).

B.11.3 Examples

Figure B.22 shows the normalized intensity $|F|^2/V^2$, computed with $R = 10$ nm, $H = 13$ nm, and $\alpha = 60^\circ$.

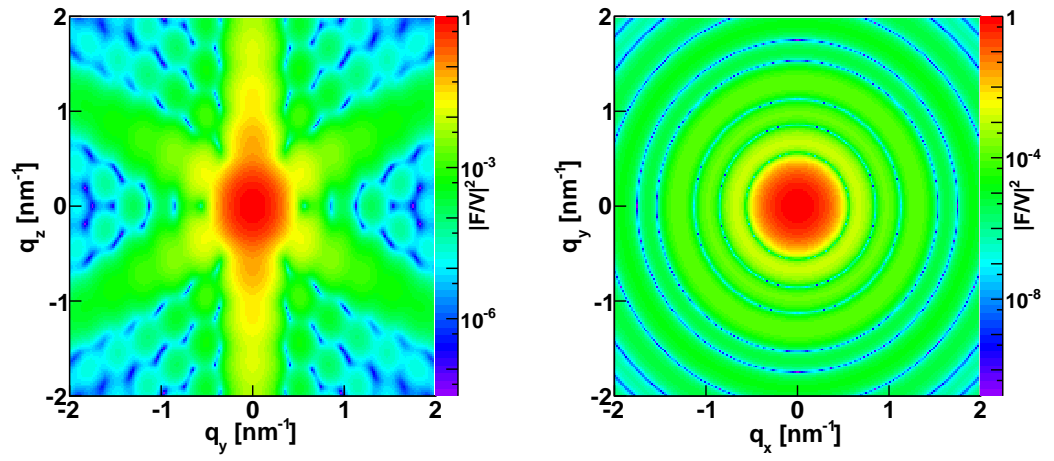


Figure B.22: Normalized intensity for the form factor of a Cone $|F|^2/V^2$, plotted against (q_z, q_y) and (q_x, q_y) . It has been computed with $R = 10$ nm, $H = 13$ nm, and $\alpha = 60^\circ$.

B.12 Full Sphere

B.12.1 Real-space geometry

The full sphere is parametrized by its radius R .

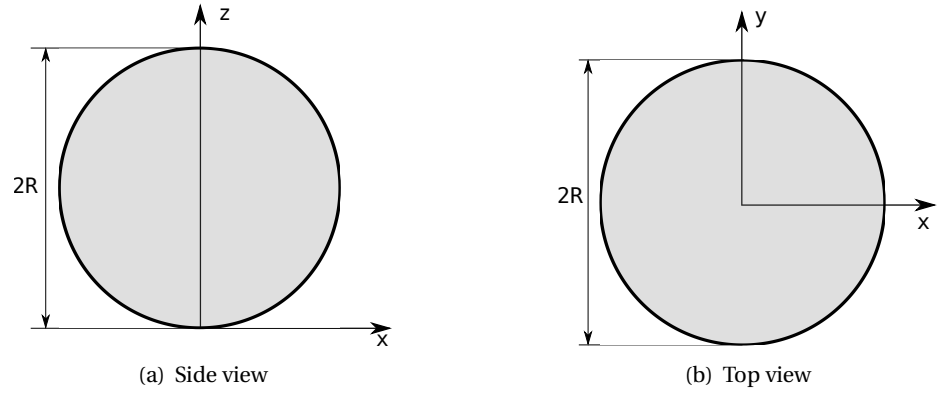


Figure B.23: Sketch of a Full Sphere.

Parameters: radius R .

Properties:

- volume $V = \frac{4\pi}{3} R^3$,
- particle surface seen from above $S = \pi R^2$.

B.12.2 Expression of the form factor

$$F_{\text{FullSphere}}(\mathbf{q}, R) = 4\pi R^3 \exp(i q_z R) \frac{\sin(qR) - qR \cos(qR)}{(qR)^3}, \quad (\text{B.2})$$

where $q = \sqrt{q_x^2 + q_y^2 + q_z^2}$.

Syntax: `FormFactorFullSphere(radius)`

B.12.3 Examples

Figure B.24 shows the normalized intensity $|F|^2/V^2$, computed with $R = 8$ nm.

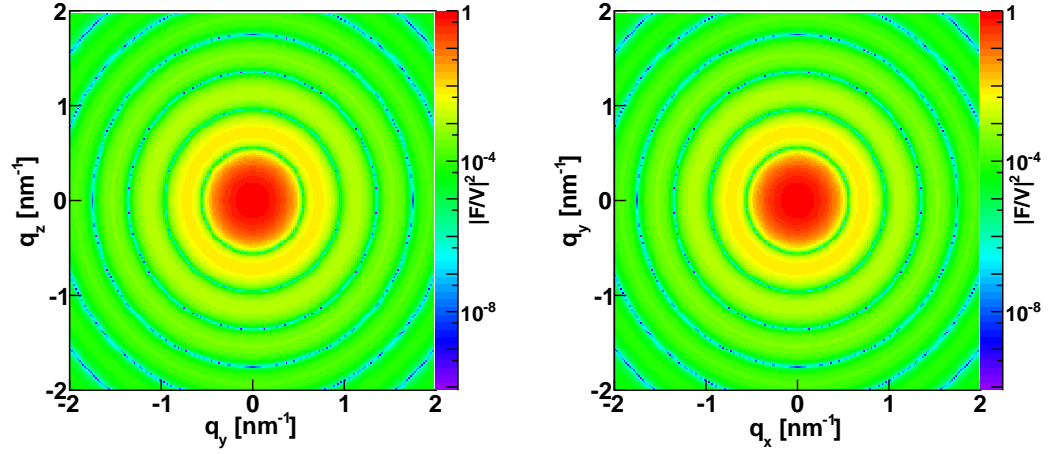


Figure B.24: Normalized intensity for the form factor of a Full Sphere $|F|^2/V^2$, plotted against (q_z, q_y) and (q_x, q_y) and computed with $R = 8$ nm.

B.13 Truncated Sphere

B.13.1 Real-space geometry

This shape is a spherical dome, *i.e.* a portion of a sphere cut off by a plane (perpendicular to z -axis) as shown in fig. B.25.

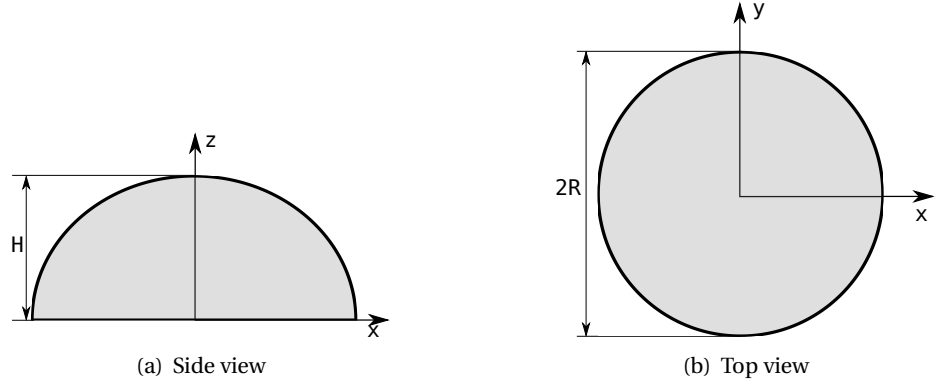


Figure B.25: Sketch of a Truncated Sphere.

Parameters:

- radius R ,
- height H .

Restrictions on the parameters: $0 \leq H \leq 2R$.

Properties:

- volume $V = \pi R^3 \left[\frac{2}{3} + \frac{H-R}{R} - \frac{1}{3} \left(\frac{H-R}{R} \right)^3 \right]$,
- particle surface seen from above $S = \begin{cases} \pi R^2, & H \geq R \\ \pi (2RH - H^2), & H < R \end{cases}$.

B.13.2 Expression of the form factor

$$F_{\text{TruncatedSphere}}(\mathbf{q}, R, H) = 2\pi \exp[iq_z(H-R)] \int_{R-H}^R R_z^2 \frac{J_1(q_{\parallel} R_z)}{q_{\parallel} R_z} \exp(iq_z z) dz,$$

with $J_1(x)$ the first order Bessel function of the first kind [9], $q_{\parallel} = \sqrt{q_x^2 + q_y^2}$, and $R_z = \sqrt{R^2 - z^2}$

Syntax: FormFactorTruncatedSphere(radius, height)

B.13.3 Examples

Figure B.26 shows the normalized intensity $|F|^2/V^2$, computed with $R = 5$ nm and $H = 7$ nm:

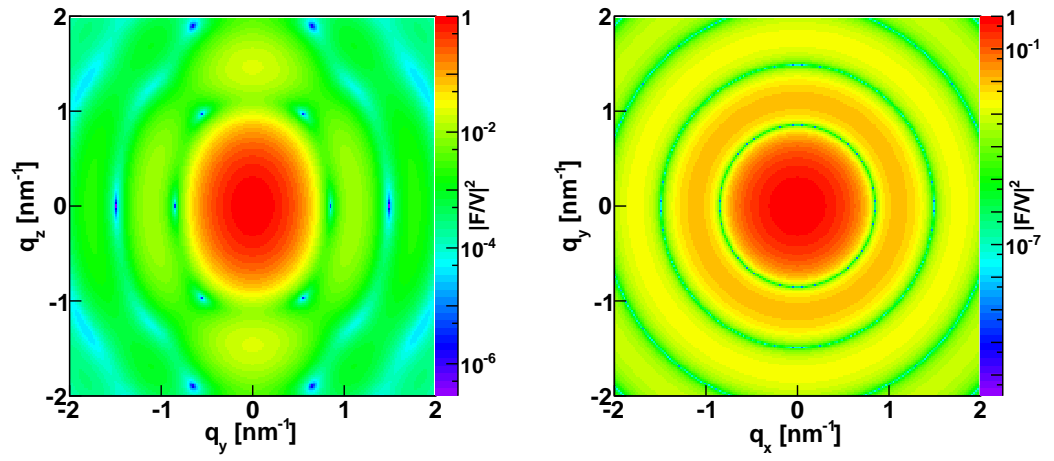


Figure B.26: Normalized intensity for the form factor of a Truncated Sphere $|F|^2/V^2$, plotted against (q_z, q_y) and (q_x, q_y) and computed with $R = 5$ nm and $H = 7$ nm.

B.14 Full Spheroid

B.14.1 Real-space geometry

A full spheroid is generated by rotating an ellipse around the vertical axis (see fig. B.27).

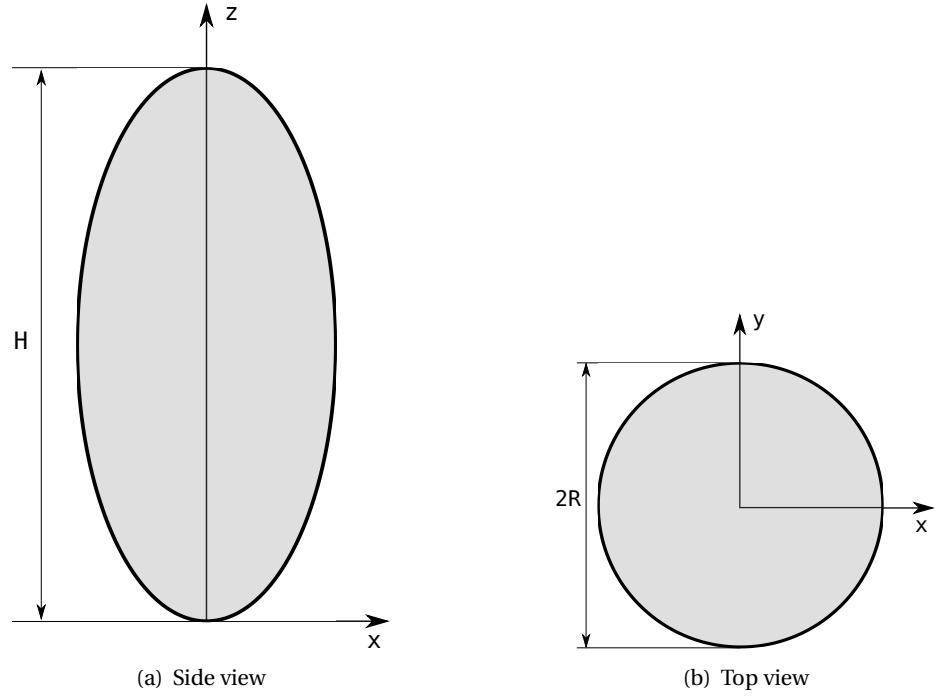


Figure B.27: Sketch of a Full Spheroid.

Parameters:

- radius R ,
- height H .

Properties:

- volume $V = \frac{2}{3} R^2 H$,
- particle surface seen from above $S = \pi R^2$.

B.14.2 Expression of the form factor

$$F_{\text{FullSpheroid}}(\mathbf{q}, R, H) = 4\pi \exp(i q_z H/2) \int_0^{H/2} R_z^2 \frac{J_1(q_{\parallel} R_z)}{q_{\parallel} R_z} \cos(q_z z) dz, \text{ with}$$

with $J_1(x)$ the first order Bessel function of the first kind [9], $R_z = R \sqrt{1 - \frac{4z^2}{H^2}}$, $\gamma_z = \sqrt{(q_x R_z)^2 + (q_y R_z)^2}$.

Syntax: FormFactorFullSpheroid(radius,height)

B.14.3 Examples

Figure B.28 shows the normalized intensity $|F|^2/V^2$, computed with $R = 10$ nm, and $H = 13$ nm.

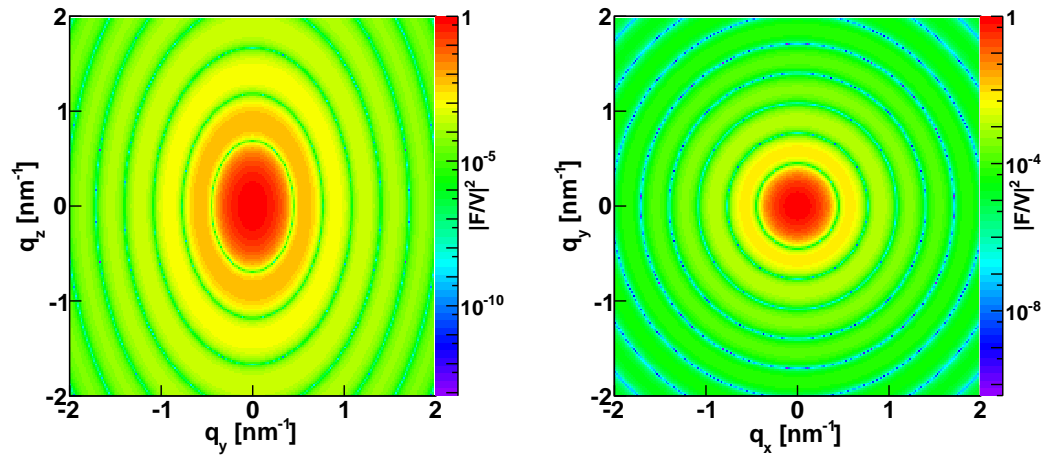


Figure B.28: Normalized intensity for the form factor of a full spheroid $|F|^2/V^2$, plotted against (q_z, q_y) and (q_x, q_y) and computed with $R = 10$ nm and $H = 13$ nm.

B.15 Truncated Spheroid

B.15.1 Real-space geometry

This shape is a spheroidal dome: a portion of a full spheroid cut off by a plane perpendicular to the z -axis.

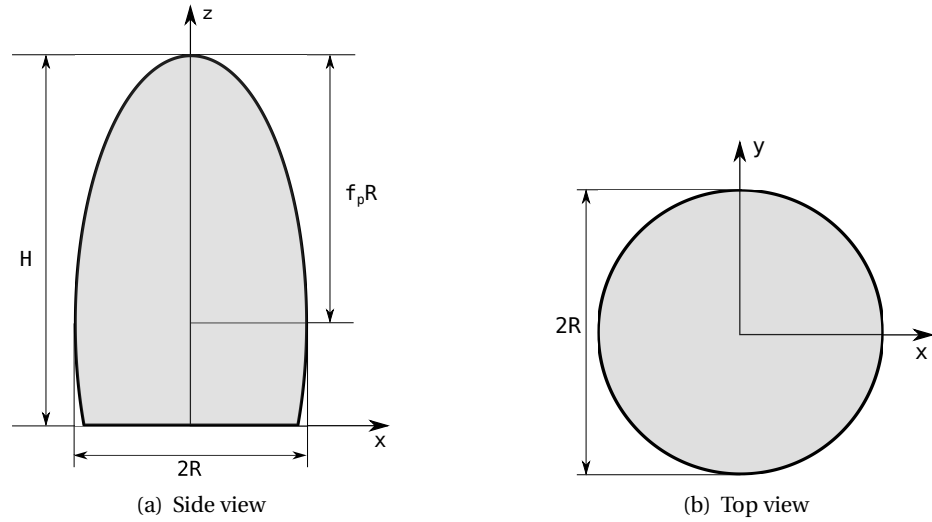


Figure B.29: Sketch of a Truncated Spheroid.

Parameters:

- radius R ,
- height H ,
- height_flattening coefficient in the perpendicular direction f_p .

Restrictions on the parameters: $0 < \frac{H}{R} < 2f_p$.

Properties:

- volume $V = \frac{\pi R H^2}{f_p} \left(1 - \frac{H}{3f_p R}\right)$,
- particle surface seen from above $S = \begin{cases} \pi R^2, & H \geq f_p R \\ \pi \left(\frac{2RH}{f_p} - \frac{H^2}{f_p^2} \right), & H < R \end{cases}$.

B.15.2 Expression of the form factor

$$F_{\text{TruncatedSpheroid}}(\mathbf{q}, R, H, f_p) = 2\pi \exp[iq_z(H - f_p R)] \int_{f_p R - H}^{f_p R} R_z^2 \frac{J_1(q_{\parallel} R_z)}{q_{\parallel} R_z} \exp(iq_z z) dz$$

with $J_1(x)$ the first order Bessel function of the first kind [9], $q_{\parallel} = \sqrt{q_x^2 + q_y^2}$ and $R_z = \sqrt{R^2 - z^2/f_p^2}$.

Syntax: `FormFactorTruncatedSpheroid(radius, height, height_flattening)`

B.15.3 Examples

Figure B.30 shows the normalized intensity $|F|^2/V^2$, computed with $R = 7.5$ nm, $H = 9$ nm and $f_p = 1.2$.

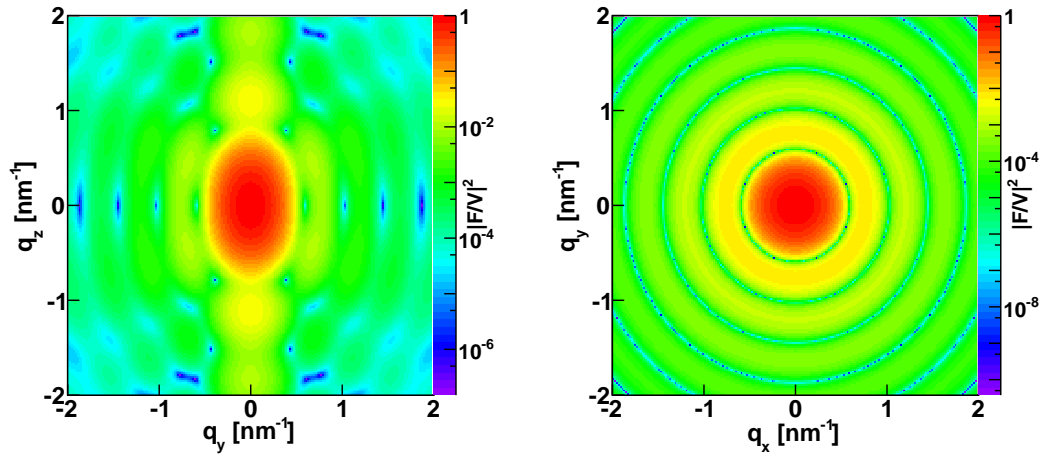


Figure B.30: Normalized intensity for the form factor of a Truncated Spheroid $|F|^2/V^2$, plotted against (q_z, q_y) and (q_x, q_y) and computed with $R = 7.5$ nm, $H = 9$ nm, and $f_p = 1.2$.

B.16 Hemi ellipsoid

B.16.1 Real-space geometry

This shape is a truncated ellipsoid as shown in fig. B.31.

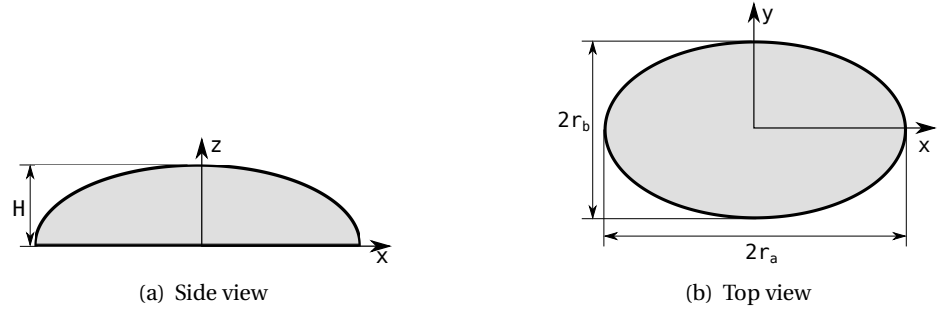


Figure B.31: Sketch of an Hemi-ellipsoid.

Parameters:

- r_a = half length of the ellipse main axis parallel to x ,
- r_b = half length of the ellipse main axis parallel to y ,
- H = height (half length of the vertical main axis of a full ellipsoid).

Properties:

- volume $V = \frac{2}{3}\pi r_a r_b H$,
- particle surface seen from above $S = \pi r_a r_b$.

B.16.2 Expression of the form factor

$$F_{\text{hemi-ellipsoid}}(\mathbf{q}, r_a, r_b, H) = 2\pi \int_0^H r_{a,z} r_{b,z} \frac{J_1(\gamma_z)}{\gamma_z} \exp(iq_z z) dz,$$

with $J_1(x)$ the first order Bessel function of the first kind [9], $r_{a,z} = r_a \sqrt{1 - \left(\frac{z}{H}\right)^2}$, $r_{b,z} = r_b \sqrt{1 - \left(\frac{z}{H}\right)^2}$ and $\gamma_z = \sqrt{(q_x r_{a,z})^2 + (q_y r_{b,z})^2}$.

Syntax: FormFactorHemiEllipsoid(r_a , r_b , height)

B.16.3 Examples

Figure B.32 shows the normalized intensity $|F|^2/V^2$, computed with $r_a = 10$ nm, $r_b = 6$ nm and $H = 8$ nm.

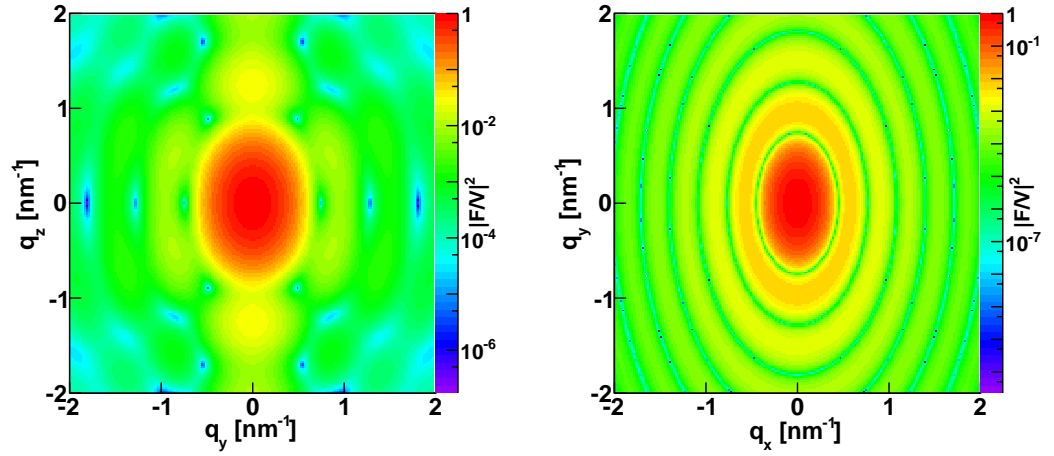


Figure B.32: Normalized intensity for the form factor of an Hemi-Ellipsoid $|F|^2/V^2$, plotted against (q_z, q_y) and (q_x, q_y) computed with $r_a = 10$ nm, $r_b = 6$ nm, and $H = 8$ nm.

B.17 Ripple1

B.17.1 Real-space geometry

This shape has a sinusoidal profile (see fig. B.33).

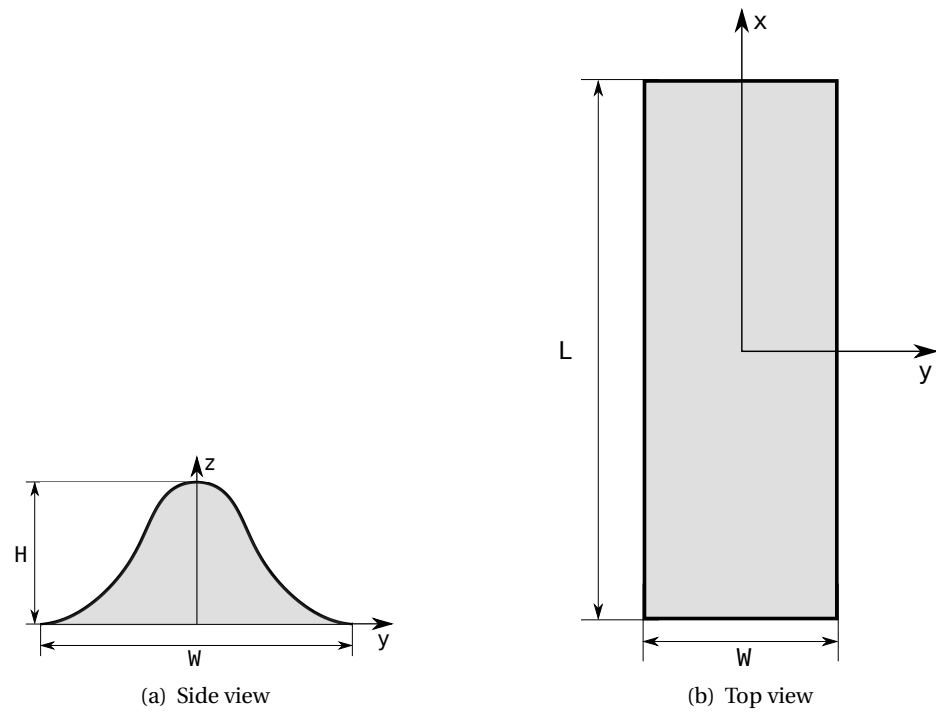


Figure B.33: Sketch of a Ripple1.

Parameters:

- length L ,
- width W ,
- height H .

Properties:

- volume $V = \frac{LWH}{2}$,
- particle surface seen from above $S = LW$.

B.17.2 Expression of the form factor

$$F_{\text{ripple1}}(\mathbf{q}, L, W, H) = L \cdot \frac{W}{\pi} \cdot \text{sinc}\left(\frac{q_x L}{2}\right) \times \int_0^H dz \arccos\left(\frac{2z}{H} - 1\right) \text{sinc}\left[\frac{q_y W}{2\pi} \arccos\left(\frac{2z}{H} - 1\right)\right] \exp(i q_z z),$$

where arccos is the arc cosine (*i.e.* the inverse operation of cosine).

Syntax: FormFactorRipple1(length, width, height)

B.17.3 Examples

Figure B.34 shows the normalized intensity $|F|^2/V^2$, computed with $L = 27$ nm, $W = 20$ nm and $H = 14$ nm.

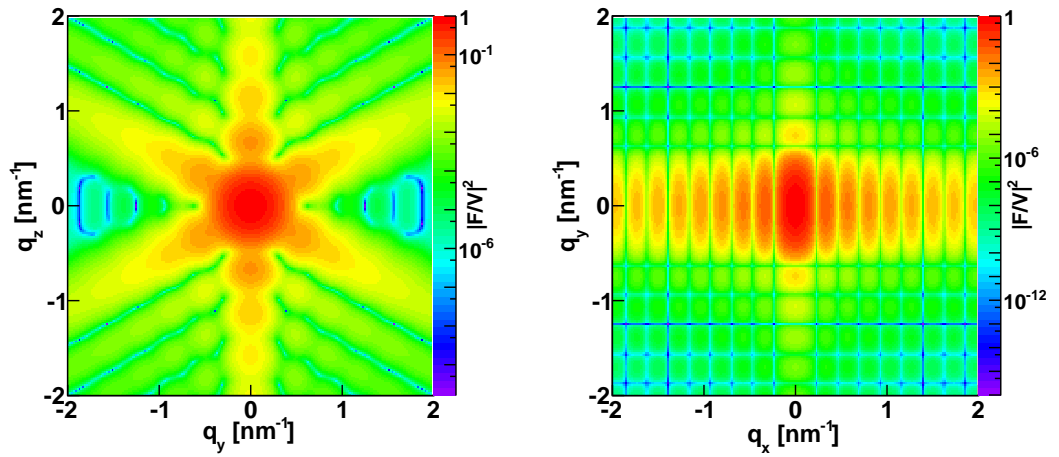


Figure B.34: Normalized intensity for the form factor of a ripple1 $|F|^2/V^2$, plotted against (q_z, q_y) and (q_x, q_y) computed with $L = 27$ nm, $W = 20$ nm, and $H = 14$ nm.

B.18 Ripple2

B.18.1 Real-space geometry

This shape has an asymmetric sawtooth profile.

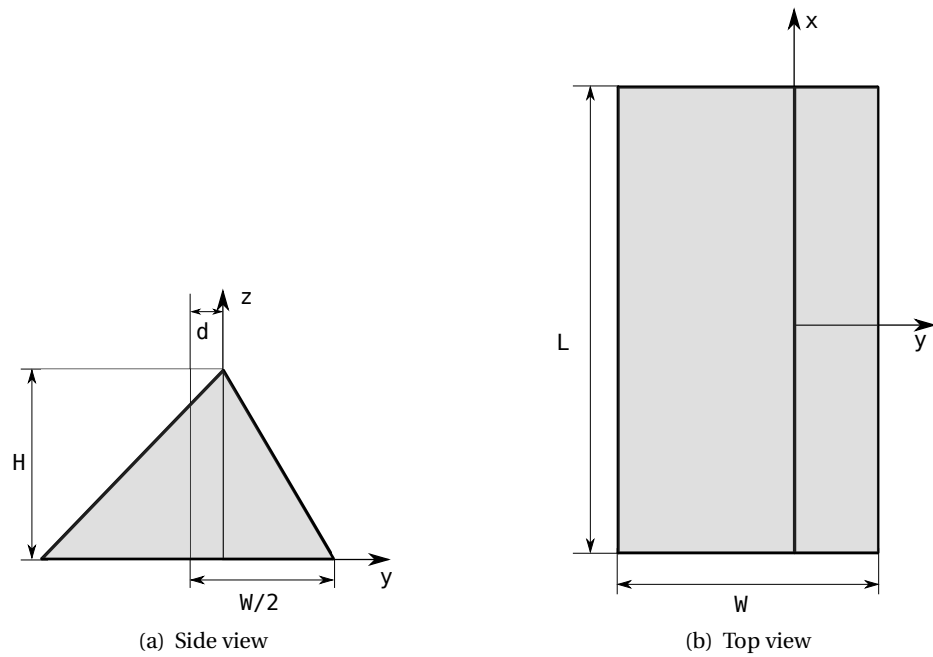


Figure B.35: Sketch of a Ripple2.

Parameters:

- length L ,
- width W ,
- height H ,
- asymmetry d .

Restriction on the parameters: $|d| < \frac{W}{2}$.

Properties:

- volume $V = \frac{LWH}{2}$,
- particle surface seen from above $S = LW$.

B.18.2 Expression of the form factor

$$F_{\text{ripple2}}(\mathbf{q}, L, W, H, d) = LW \operatorname{sinc}\left(\frac{q_x L}{2}\right) \times \int_0^H \left(1 - \frac{z}{H}\right) \operatorname{sinc}\left[\frac{q_y W}{2} \left(1 - \frac{z}{H}\right)\right] \exp\left\{i \left[q_z z - q_y d \left(1 - \frac{z}{H}\right)\right]\right\} dz$$

Syntax: FormFactorRipple2(length, width, height, asymmetry)

B.18.3 Examples

Figure B.36 shows the normalized intensity $|F|^2/V^2$, computed with $L = 36$ nm, $W = 25$ nm, $H = 14$ nm, and $d = 3$ nm.

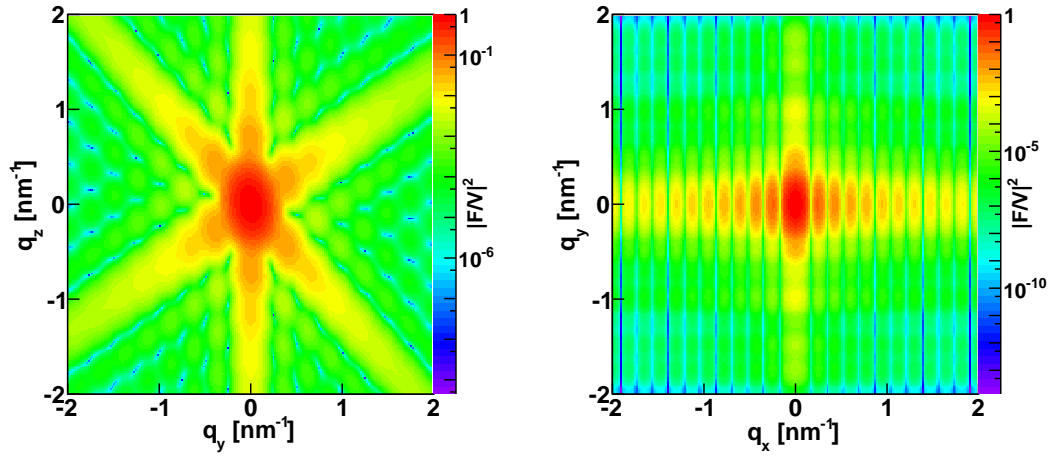


Figure B.36: Normalized intensity for the form factor of a ripple2 $|F|^2/V^2$, plotted against (q_z, q_y) and (q_x, q_y) computed with $L = 36$ nm, $W = 25$ nm, $H = 14$ nm, and $d = 3$ nm.

B.19 Distorted Wave Born Approximation

The previous sections of this appendix on form factors have dealt with the Born approximation. In this case the form factor is given by a single integral over the particle shape (see equation B.1). But this approximation fails when multiple reflections and refractions have to be taken into account at interfaces because of the presence of underlying layers of materials and the closeness of the incident angle α_i to the critical angle of total external reflection α_c . The first order correction to the scattering theory is the Distorted Wave Born Approximation (DWBA), whereas the Born approximation is the zeroth order.

The collective effects between the particles are not considered in this section. They will be dealt with in... We also do not take any polarization effects into account.

In the DWBA, the form factor of a particle in a multilayer system is given by

$$F_{\text{DWBA}}(\mathbf{k}_i, \mathbf{k}_f, r_z) = T_i T_f F_{\text{BA}}(\mathbf{k}_i - \mathbf{k}_f) e^{i(k_{i,z} - k_{f,z})r_z} + R_i T_f F_{\text{BA}}(\tilde{\mathbf{k}}_i - \mathbf{k}_f) e^{i(-k_{i,z} - k_{f,z})r_z} \\ + T_i R_f F_{\text{BA}}(\mathbf{k}_i - \tilde{\mathbf{k}}_f) e^{i(k_{i,z} + k_{f,z})r_z} + R_i R_f F_{\text{BA}}(\tilde{\mathbf{k}}_i - \tilde{\mathbf{k}}_f) e^{i(-k_{i,z} + k_{f,z})r_z}, \quad (\text{B.3})$$

where F_{BA} is the expression of the form factor in the Born approximation, r_z is the z -coordinate of the particle's position, $\mathbf{k}_i = (k_{i,x}, k_{i,y}, k_{i,z})$ $\mathbf{k}_f = (k_{f,x}, k_{f,y}, k_{f,z})$ are the incident and scattered wave vectors in air, respectively. With a tilde ($\tilde{}$), these wavevectors components are evaluated in the multilayer system (the refractive indices of the different constituting materials have to be taken into account). T_i, T_f, R_i, R_f are the transmission and reflection coefficients for the incident wave (index i) or the scattered one (index f). These coefficients can be calculated using the Parratt formalism [10] or the matrix method [11]. $\mathbf{k}_i - \mathbf{k}_f$ is equal to the scattering vector \mathbf{q} and the z -axis is pointing upwards.



Remark: The particles cannot sit in between layers. At most they can be sitting on any inner interfaces.

In the followings, the DWBA will be illustrated for two different layouts of particles:

- particles deposited on a substrate,
- particles buried in a layer on a substrate.

B.19.1 Particles deposited on a substrate

In this configuration, the particles are sitting on top of a substrate layer, in the air as shown in fig. B.37. In the DWBA the expression of a form factor becomes

$$F_{\text{DWBA}}(q_{\parallel}, k_{i,z}, k_{f,z}) = F_{\text{BA}}(q_{\parallel}, k_{i,z} - k_{f,z}) + R_i F_{\text{BA}}(q_{\parallel}, -k_{i,z} - k_{f,z}) \\ + R_f F_{\text{BA}}(q_{\parallel}, k_{i,z} + k_{f,z}) + R_i R_f F_{\text{BA}}(q_{\parallel}, -k_{i,z} + k_{f,z}), \quad (\text{B.4})$$

where q_{\parallel} is the component of the scattering beam in the plane of the interface ($\mathbf{q} = \mathbf{k}_i - \mathbf{k}_f$), $k_{i,z}$ and $k_{f,z}$ are the z -component of the incident and scattered beam, respectively. $R_i,$

R_f are the reflection coefficients in incidence and reflection. They are defined as $R = \frac{k_z + \sqrt{n_s^2 k_0^2 - |k_{\parallel}|^2}}{k_z - \sqrt{n_s^2 k_0^2 - |k_{\parallel}|^2}}$, where $n_s = 1 - \delta_s - i\beta_s$ is the refractive index of the substrate, k_0 is the wavelength in vacuum ($2\pi/\lambda$), k_z and k_{\parallel} are the z -component and the in-plane component of \mathbf{k}_i or \mathbf{k}_f .



Remark: If the particles are sitting on a multilayered system, the expression of the form factor in the DWBA is obtained by replacing the Fresnel coefficient by the corresponding coefficients of the underlying layers [10, 11].

Figure B.37 illustrates the four scattering processes for a supported particle, taken into account in the DWBA. The first term of eq. B.4 corresponds to the Born approximation. Each term is weighted by a Fresnel coefficient.

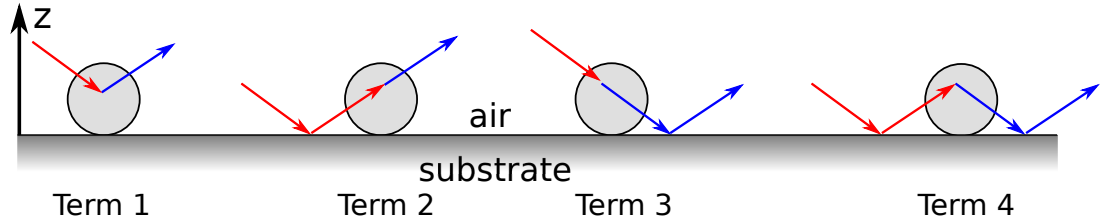


Figure B.37: Schematic views of the different terms appearing in the expression of the form factor under DWBA for particles sitting on a substrate layer.

Script B.1 illustrates the difference between BA and DWBA in BornAgain and figure B.38 shows the intensity contourplot generated using this script with (full) spheroids as particles. This script considers the simple case of:

- one kind of particles' shape,
- no interference between the particles,
- in the DWBA, a sample made of a layer of substrate on which are deposited the particles,
- in the BA, a sample composed of the particles in air.

Listing B.1: Python script to generate figure B.38. The difference between BA and DWBA in this simple case is the absence or presence of a substrate layer in the sample.

```
import numpy
import matplotlib
import pylab
from libBornAgainCore import *
```



```

def get_sample():
    """
    Build and return the sample to calculate formfactor in Born
    or Distorted Wave Born Approximation.
    """
    # defining materials
    m_ambience = MaterialManager.getHomogeneousMaterial("Air",
        0.0, 0.0)
    m_substrate = MaterialManager.getHomogeneousMaterial("
        Substrate", 6e-6, 2e-8)
    m_particle = MaterialManager.getHomogeneousMaterial("Particle
        ", 6e-4, 2e-8)

    # collection of particles
    ff= FormFactorSpheroid(7.5*nanometer, 9.0*nanometer, 1.2)
    particleshape = Particle(m_particle, ff)
    particle_decoration = ParticleDecoration()
    particle_decoration.addParticle(particleshape, 0.0, 1.0)
    interference = InterferenceFunctionNone()
    particle_decoration.addInterferenceFunction(interference)
    air_layer = Layer(m_ambience)
    air_layer.setDecoration(particle_decoration)
    substrate_layer = Layer(m_substrate, 0)

    # Sample = particles in air for BA or particles in air and
    # sitting on a substrate for DWBA
    multi_layer = MultiLayer()
    multi_layer.addLayer(air_layer)
    # Add substrate layer for DWBA
    # Comment the following line out for BA
    multi_layer.addLayer(substrate_layer)
    return multi_layer

def get_simulation():
    """
    Create and return GISAXS simulation with beam and detector
    defined
    """
    simulation = Simulation()
    simulation.setDetectorParameters(200, 0.0*degree, 2.0*degree,
        200, 0.0*degree, 2.0*degree, True)
    simulation.setBeamParameters(1.0*angstrom, 0.5*degree, 0.0*
        degree)
    return simulation

def run_simulation():
    """
    Run simulation and plot results

```

```

"""
sample = get_sample()
simulation = get_simulation()
simulation.setSample(sample)
simulation.runSimulation()
pylab.imshow(numpy.rot90(result, 1), norm=matplotlib.colors.
              LogNorm(), extent=[0.0, 2.0, 0, 2.0])
pylab.show()

if __name__ == '__main__':
    run_simulation()

```

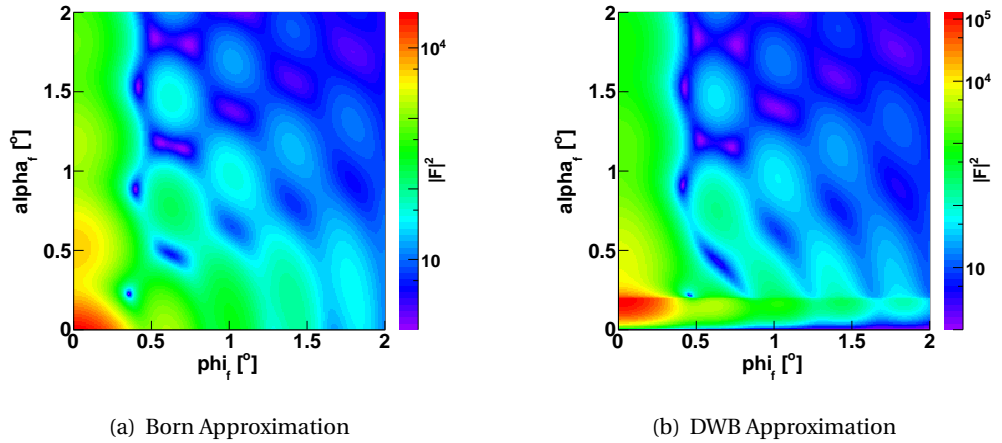


Figure B.38: Intensity map of Spheroid form factor in BA and DWBA computing using script B.1 and FormFactorSpheroid(7.5*nanometer, 9.0*nanometer, 1.2).



Remark: In BornAgain, the DWBA is implemented automatically when assembling the sample with more than the air layer.

B.19.2 Buried particles

The system considered in this section consists of particles encapsulated in a layer, which is sitting on a substrate (see fig. B.39). In this case the form factor in the DWBA is given by

$$\begin{aligned}
 F_{\text{DWBA}}(q_{\parallel}, k_{i,z}, k_{f,z}) = & T_i T_f F_{\text{BA}}(q_{\parallel}, k_{i,z} - k_{f,z}) e^{i(k_{i,z} - k_{f,z})d} + R_i T_f F_{\text{BA}}(q_{\parallel}, -k_{i,z} - k_{f,z}) e^{i(-k_{i,z} - k_{f,z})d} \\
 & + R_f T_i F_{\text{BA}}(q_{\parallel}, k_{i,z} + k_{f,z}) e^{i(k_{i,z} + k_{f,z})d} + R_f R_i F_{\text{BA}}(q_{\parallel}, -k_{i,z} + k_{f,z}) e^{i(-k_{i,z} + k_{f,z})d},
 \end{aligned}
 \quad (\text{B.5})$$

$$R_j = \frac{t_{0,1}^j r_{1,2}^j \exp(2ik_{j,z}t)}{1 + r_{0,1}^j r_{1,2}^j \exp(2ik_{j,z}t)}, \quad T_j = \frac{t_{0,1}^j}{1 + r_{0,1}^j r_{1,2}^j \exp(2ik_{j,z}t)}, \quad j = i, f$$

where q_{\parallel} is the component of the scattering beam in the plane of the interface, $k_{i,z}$ and $k_{f,z}$ are the z -component of the incident and scattered beams, respectively. d is the depth at which the particles are sitting in the layer. Note that this value is given relative to the top of this layer and it is not the coordinate in the absolute referential (linked with the full sample) and it is measured up to the bottom of the particle. t is the thickness of the intermediate layer containing the particles. $R_{i,f}$ and $T_{i,f}$ are the reflection and transmission coefficients in incidence and reflection (they can be calculated using Parratt or matrix formalism). $r_{0,1}^j$, $r_{1,2}^j$, $t_{0,1}^j$ are the reflection and transmission coefficients between layers; the indices are related to different boundaries with 0: air, 1: intermediate layer and 2: substrate layer and the superscript j is associated with the incident or scattered beams:

$$r_{n,n+1}^j = \frac{k_{j,z,n} - k_{j,z,n+1}}{k_{j,z,n} + k_{j,z,n+1}}, \quad t_{n,n+1}^j = \frac{2k_{j,z,n}}{k_{j,z,n} + k_{j,z,n+1}}, \quad n = 0, 1, \quad j = i, f,$$

where n is related to the layers, z to the vertical component, and j to the beams.

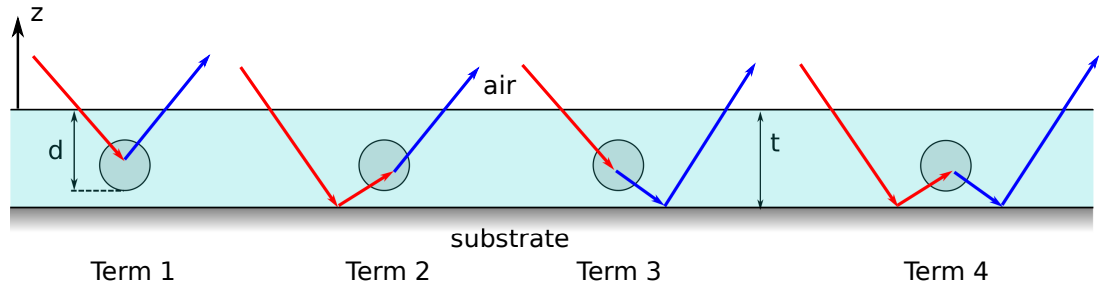


Figure B.39: Schematic views of the different terms appearing in the expression of the form factor under the DWBA for buried particles.

Figure B.40 shows a typical example of the output intensity scattered from a sample made of 3 layers: air, substrate, and in between, spherical particles embedded in the middle of a 30 nm-thick layer. This figure had been generated using listing B.2.

Listing B.2: Python script to generate fig.B.40. Spherical particles are embedded in the middle of a layer on a substrate.

```
import numpy
import matplotlib
import pylab
from libBornAgainCore import *

def get_sample():
    """
    Build and return the sample in Distorted Wave Born
    Approximation.
    """
```

```

# defining materials
m_ambience = MaterialManager.getHomogeneousMaterial("Air",
    0.0, 0.0)
m_interm_layer = MaterialManager.getHomogeneousMaterial("
    InterMLayer", 3.45e-6, 5.24e-9)
m_substrate = MaterialManager.getHomogeneousMaterial("
    Substrate", 7.43e-6, 1.72e-7)
m_particle = MaterialManager.getHomogeneousMaterial("Particle
    ", 0.0, 0.0)

# collection of particles
ff = FormFactorFullSphere(10.2*nanometer)
particleshape = Particle(m_particle, ff)
particle_decoration = ParticleDecoration()
particle_decoration.addParticle(particleshape, 20.1, 1.0)
interference = InterferenceFunctionNone()
particle_decoration.addInterferenceFunction(interference)
air_layer = Layer(m_ambience)
intermediate_layer = Layer(m_interm_layer, 30.*nanometer)
intermediate_layer.setDecoration(particle_decoration)
substrate_layer = Layer(m_substrate, 0)

multi_layer = MultiLayer()
multi_layer.addLayer(air_layer)
multi_layer.addLayer(intermediate_layer)
multi_layer.addLayer(substrate_layer)
return multi_layer

def get_simulation():
    """
    Create and return GISAXS simulation with beam and detector
    defined
    """
    simulation = Simulation()
    simulation.setDetectorParameters(400, 0., 1.*degree, 400, 0.,
        1.*degree, True)
    simulation.setBeamParameters(1.5*angstrom, 0.15*degree, 0.)
    return simulation

def run_simulation():
    """
    Run simulation and plot results
    """
    sample = get_sample()
    simulation = get_simulation()
    simulation.setSample(sample)
    simulation.runSimulation()
    result = simulation.getIntensityData().getArray() + 1 # for

```

```

log scale

pylab.imshow(numpy.rot90(result, 1), norm=matplotlib.colors.
    LogNorm(), extent=[0.0, 1.0, 0, 1.0])
pylab.show()

if __name__ == '__main__':
    run_simulation()

```

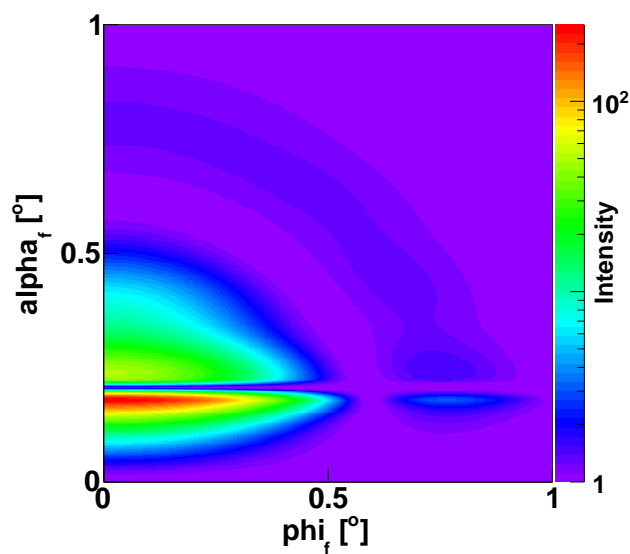


Figure B.40: Map of intensity scattered from a sample made of spherical particles embedded in the middle of a 30 nm-thick layer on a substrate (see Script B.2 for details).



Remark: For layers different from the air layer, the top interface is considered as the reference level to position the encapsulated particles. For example, spheres positioned at depth d (positive) are located at a distance d from the top of the layer up to the bottom of these particles. This convention is different for the top air layer, where particles sitting at the interface with an underlying layer (*i.e.* the bottom of the air layer) are located at depth 0 (see fig. B.41).

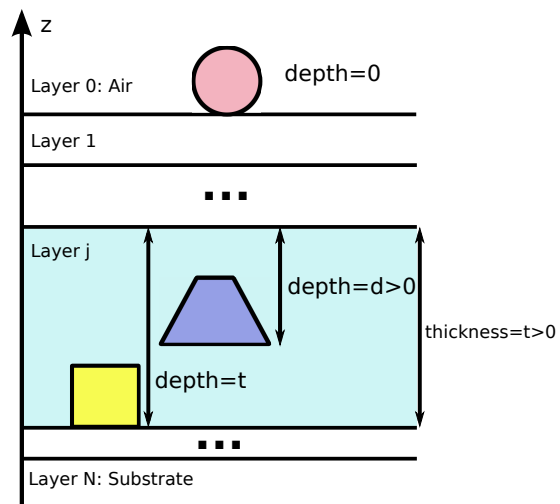


Figure B.41: Illustration of the convention about depth used in BornAgain to encapsulate particles in layers.

B.20 Core-shell particles

BornAgain also offers the possibility to simulate more complicated shapes of particles by combining those listed in the previous sections. To generate a core-shell particle, the combination is performed using the following command:

`ParticleCoreShell(shell_particle, core_particle, relative_core_position)`, where `shell_particle` and `core_particle` are the outer and inner parts of the core-shell particle, respectively. They refer to one of the form factors defined previously and to an associated material. For example, for the outer part, `shell_particle=Particle(material_shell, outer_form_factor)`, where `material_shell` is the material of the shell and `outer_form_factor` is the shape of the outer part (cf. listing B.3).

`relative_core_position` defines the position of the centre of gravity of the inner shape with respect to the outer one. An example in fig. B.42 shows a core shell particle made of a box for the outer part and of a shifted pyramidal shape for the inner one.

Figure B.43 displays the output intensity scattered in the Born Approximation using the code listed in B.3 to generate the sample, and the incident angles $\alpha_i = 0.2^\circ$ and $\phi_i = 0^\circ$.

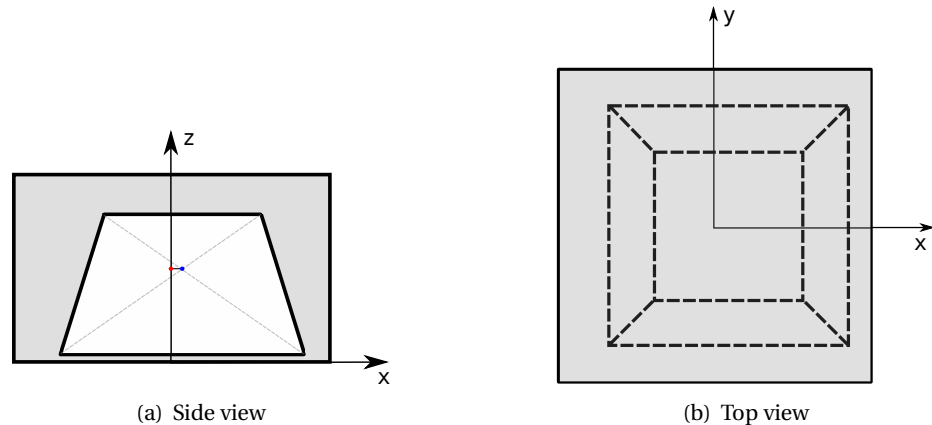


Figure B.42: Example of a core-shell particle composed of a box with a pyramidal inset. The relative core shell position is marked by the position of its center of gravity (blue point •) with respect to the center of gravity of the box (red point •).

Listing B.3: Python script to create a core-shell particle made of a box with a pyramidal shifted inset.

```
def get_sample():
    """
    Build and return the sample to calculate core-shell
    formfactor in Born Approximation.
    """
    # defining materials
```

```
m_air = MaterialManager.getHomogeneousMaterial("Air", 0.0,
0.0)
m_shell = MaterialManager.getHomogeneousMaterial("Shell", 1e
-4, 2e-8)
m_core = MaterialManager.getHomogeneousMaterial("Core", 6e-5,
2e-8)

# collection of particles
outer_ff = FormFactorBox(16.0*nanometer, 16.0*nanometer, 8.0*
nanometer)
inner_ff = FormFactorPyramid(12.0*nanometer, 7.0*nanometer,
60.0*degree)
shell_particle = Particle(m_shell, outer_ff)
core_particle = Particle(m_core, inner_ff)
core_position = kvector_t(1.5, 0.0, 0.0)

particle = ParticleCoreShell(shell_particle, core_particle,
core_position)
particle_decoration= ParticleDecoration()
particle_decoration.addParticle(particle)
interference = InterferenceFunctionNone()
particle_decoration.addInterferenceFunction(interference)

air_layer = Layer(m_air)
air_layer.setDecoration(particle_decoration)

multi_layer = MultiLayer()
multi_layer.addLayer(air_layer)

return multi_layer
```

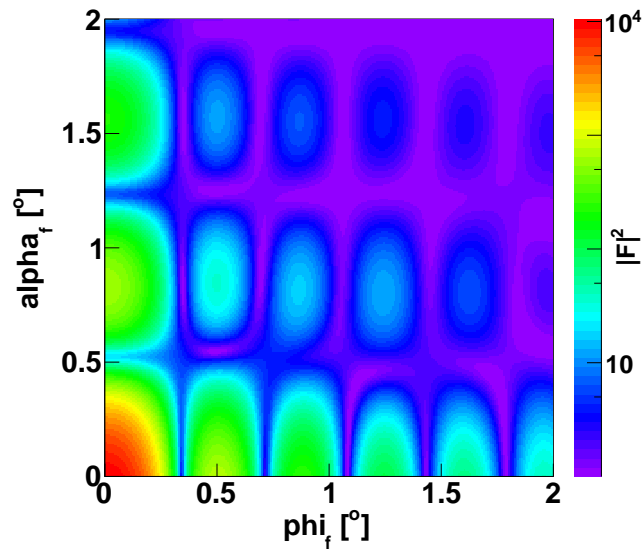



Figure B.43: Intensity map of a core-shell form factor in Born Approximation using `FormFactorBox(16*nanometer, 16*nanometer, 8*nanometer)` and `FormFactorPyramid(12*nanometer, 7*nanometer, 60*degree)` for the outer and inner shells, respectively. The core particle is shifted by 1.5 nm in the x -direction with respect to the centre of the outer shell. The sample used to generate this figure is listed in B.3. There is no substrate and no interference between the particles.

Appendix C

Collection of particles

In the section we are going to describe how to simulate collections of particles using *BornAgain* *i.e.* the way their spatial distributions and the distribution of shapes and their correlations can influence the output scattered intensity. The samples generated with *BornAgain* are made of different material layers characterized by their thicknesses, refractive indices, and possible surface roughnesses. Except for the thickness, the other dimensions of the layers are infinite. Particles can be embedded or deposited on the top of any layers. Those particles are characterized by their shapes, refractive indices, their spatial distribution and concentration in the sample. When the particles are densely packed, the distance relative to each other becomes of the same order as the particles' sizes. The radiation scattered from these various particles are going to interfere together. The influence of the particles' shapes has been described in the previous section about form factors.

We do not consider any multiple scattering, polarisation (see Section...), nor layers' roughness (see Section...).

We are first going to give a short overview of the theory involved, mostly in order to define the terminology. For a more complete theoretical description, the user is referred to, for example, [12]. Then we are going to describe how the interference features have been implemented in *BornAgain* and give some detailed examples.

C.1 Theory

Considering a set of N particles labeled with index i , located at \mathbf{R}_i and having shapes $S_i(\mathbf{r})$ ($S_i = 0$ outside the particle and 1 inside), the scattered intensity per particle is given by:

$$I(\mathbf{q}) = \frac{1}{N} \left\langle \left| \sum_i F_i(\mathbf{q}) e^{i\mathbf{q} \cdot \mathbf{R}_i} \right|^2 \right\rangle = \frac{1}{N} \left\langle \sum_i |F_i(\mathbf{q})|^2 + \sum_{i \neq j} F_i(\mathbf{q}) F_j^*(\mathbf{q}) \exp(i\mathbf{q} \cdot (\mathbf{R}_i - \mathbf{R}_j)) \right\rangle, \quad (\text{C.1})$$

where $\langle \dots \rangle$ denotes a spatial and temporal average, \mathbf{q} is the wave vector (reciprocal space) and F_i is the form factor of particle i evaluated using the Distorted Wave Born Approximation.

If only the statistical quantities of the system are known (particles' positions and sizes), the discrete sums in equation C.1 can be replaced by continuous integrals using some prob-

ability densities. For example, in two dimensions (which is the case for particles deposited on a surface), the probability per unit surface to find a particle of class α in \mathbf{R}_i knowing that there is a particle of type β in \mathbf{R}_j can be written as $\rho_S^2 g_{\alpha\beta}(R_{i,\alpha}, R_{j,\beta})$ where ρ_S is the number of particles per unit surface and $g_{\alpha\beta}$ is the partial pair correlation function, which tends towards 1 as the distance between the particles increases.

Terminology



For collections of particles, the scattered intensity contains contributions from neighboring particles. This additional pattern can be called the structure factor, the interference function or even in crystallography, the lattice factor. In this manual, we use the term "interference function" or interferences.

C.1.1 Size-distribution models

To proceed further, when the morphology and topology are not exactly known, some hypotheses needs to be made since the correlation between the kinds of scatterers and their relative positions included in $g_{\alpha\beta}$ are difficult to estimate. Several options are available:

Decoupling approximation (DA) neglects all correlations. It supposes that the particles are positioned in a way that is completely independent on their kinds (shapes, sizes). Thus the kind of scattering objects and their positions are not correlated. This leads to the following expression of the scattered intensity:

$$I(\mathbf{q}) = |\langle F(\mathbf{q}) \rangle|^2 S(\mathbf{q}) + \underbrace{\langle |F(\mathbf{q})|^2 \rangle - |\langle F(\mathbf{q}) \rangle|^2}_{\text{incoherent term}}$$

where $S(\mathbf{q})$ is the total interference function (*i.e.* the Fourier transform of the particle position autocorrelation function).

In concentrated systems, DA breaks down because of correlations. One solution is to reintroduce some correlations between particles sizes and distributions (using for example the Size spacing correlation approximation described below).

Local monodisperse approximation (LMA) partially accounts for some coupling between the positions and the kinds of the particles [13]. It requires a subdivision of the layers of particles into monodisperse domains. The contributions of these subdomains are then incoherently summed up and weighted by the size-shape probabilities. In this approximation, a particle is supposed to be surrounded by particles of the same size and shape, within the coherence length of the input beam. The scattered intensity is expressed as

$$I(\mathbf{q}) = \langle |F(\mathbf{q})|^2 S(\mathbf{q}) \rangle$$

One has to remember that in most cases, this approximation corresponds to an unphysical description of the investigated systems.

DA and LMA separate the contributions of the form factors and of the interference function. For disordered systems DA and LMA give the same result as the scattering vector gets larger *i.e.* the scattered intensity is dominated by the contribution of the form factor.

Size spacing correlation approximation (SSCA) introduces correlations between poly-disperse particles and is derived from the paracrystal model (see description below and [14]).

C.1.2 Layout of particles



Remark: The particles are positioned in the same vertical layer.

The uncorrelated or disordered lattice

For very diluted distributions of particles, the particles are too far apart from each other to lead to any interference between the waves scattered by each of them. In this case the interference function is equal to 1. The scattered intensity is then entirely determined by the form factors of the particles distributed in the sample.

The regular lattice

The particles are positioned at regular intervals generating a layout characterised by its base vectors **a** and **b** (in direct space) and the angle between these two vectors. This lattice can be two or one-dimensional depending on the characteristics of the particles. For example when they are infinitely long, the implementation can be simplified and reduced to a "pseudo" 1D system.

The ideal paracrystal

A paracrystal, whose notion was developed by Hosemann[15], allows fluctuations of the lengths and orientations of lattice vectors. Paracrystals can be defined as distorted crystals in which the crystalline order has not disappeared and for which the behavior of the interference functions at small angles is coherent. It is a transition between the regular lattice and the disordered state.

For example, in one dimension, a paracrystal is generated using the following method: we place a particle at the origin. The second one is put at a distance x with a density probability $p(x)$ that is peaked at a mean value D : $\int_{-\infty}^{\infty} p(x) dx = 1$ and $\int_{-\infty}^{\infty} xp(x) dx = D$. The third one is added at a distance y from the second site using the same rule with a density probability $p_2(y) = \int_{-\infty}^{\infty} p(x)p(y-x) dx = p \otimes p(y)$.

With such a method, the pair correlation function $g(x)$ is built step by step. Its expression and the one of its Fourier transform, which is the interference function are

$$g(x) = \delta(x) + p(x) + p(x) \otimes p(x) + \dots + p(-x) + \dots \text{ and } S(q) = \text{Re} \left(\frac{1 + P(q)}{1 - P(q)} \right),$$

where $P(q)$ is the Fourier transform of the density probability $p(x)$.

In two dimensions, the paracrystal is constructed on a pseudo-regular lattice with base vectors **a** and **b** using the following conditions for the densities of probabilities:

$$\int p_{\mathbf{a}}(\mathbf{r}) d^2\mathbf{r} = \int p_{\mathbf{b}}(\mathbf{r}) d^2\mathbf{r} = 1, \int \mathbf{a} p_{\mathbf{a}}(\mathbf{r}) d^2\mathbf{r} = \mathbf{a}, \int \mathbf{b} p_{\mathbf{b}}(\mathbf{r}) d^2\mathbf{r} = \mathbf{b}.$$

Function	One dimension	Two dimensions
Cauchy	$(1 + q^2 \omega^2)^{-3/2}$	$(1 + q_x^2 c l_x^2 + q_y^2 c l_y^2)^{-3/2}$
Gauss	$\frac{1}{2} \exp(-\frac{q^2 \omega^2}{4})$	$\frac{1}{2} \exp\left(-\frac{q_x^2 c l_x^2 + q_y^2 c l_y^2}{4}\right)$
Voigt	$\frac{\eta}{2} \exp\left(-\frac{q^2 \omega^2}{4}\right) + \frac{1-\eta}{(1 + q^2 \omega^2)^{3/2}}$	$\frac{\eta}{2} \exp\left(-\frac{q_x^2 c l_x^2 + q_y^2 c l_y^2}{4}\right) + \frac{1-\eta}{(1 + q_x^2 c l_x^2 + q_y^2 c l_y^2)^{3/2}}$

Table C.1: List of probability distribution functions in reciprocal space. ω , cl stand for coherence lengths and η is a weighting coefficient.

In the ideal case the two axes are decoupled and each unit cell should retain a parallelogram shape. The interference function is given by $S(q_{\parallel}) = \prod_{k=a,b} \text{Re} \left(\frac{1 + P_k(q_{\parallel})}{1 - P_k(q_{\parallel})} \right)$ with P_k the Fourier transform of p_k , $k = a, b$.

Probability distributions

The scattering by an ordered lattice gives rise to a series of Bragg peaks situated at the nodes of the reciprocal lattice defined. Any divergence from the ideal crystalline case modifies the output spectrum by, for example, widening or attenuating the Bragg peaks. The influence of these "defects" can be accounted for in direct space using correlation functions or by truncating the lattice or, in reciprocal space with structure factors or interference functions by convoluting the scattered pics with a function which could reproduce the experimental shapes. The later option has been implemented in BornAgain. The Fourier transforms of the probability distribution functions in 1 and 2D are listed in Table C.1. They are used in 1 and 2D lattices, and 2D paracrystals.

The Cauchy distribution corresponds to $\exp(-r)$ in real space and the Voigt one is a linear combination of the Gaussian and Cauchy probability distribution functions.

C.2 Implementation in BornAgain

C.2.1 Size-distribution models

The decoupled approximation, local monodisperse approximation and size spacing correlation approximation can be used in BornAgain. The selection is made using function `SimulationParameters()` when defining the characteristics of the simulation. For example,

```
simulation = Simulation()
....
sim_params = SimulationParameters()
# interference approx chosen between: DA (default), LMA and SS
sim_params.me_if_approx = SimulationParameters.LMA
simulation.setSimulationParameters(sim_params)
```

The users can refer to Script C.5 for a more detailed implementation. By default, the decoupled approximation (DA) is used.

C.2.2 Probability distribution functions

The expressions in the reciprocal space are given in Table C.1.

One dimension

- `FTDistribution1DCauchy(ω)`,
- `FTDistribution1DGauss(ω)`,
- `FTDistribution1DVoigt(ω, η)`.

where ω is the coherence length and η is a weighting factor.

Two dimensions

- `FTDistribution2DCauchy(cl_x, cl_y)`,
- `FTDistribution2DGauss(cl_x, cl_y)`,
- `FTDistribution2DVoigt(cl_x, cl_y)`

where $cl_{x,y}$ are the coherence lengths in the x or y direction, respectively.

These functions can be used with all interference functions except the case without any interference and the one dimensional paracrystal, for which only the Gaussian case has already been implemented.

C.2.3 Interferences

The interference function is specified when building the sample. It is linked with the particles (shape, material). Examples of implementation are given at the end of each description.

Syntax: `particle_layout.addInterferenceFunction(interference_function)`, where `particle_layout` holds the information about the different shapes and their proportions for a given layer of particles, and `interference_function` is one of the following expressions:

- `InterferenceFunctionNone()`
- `InterferenceFunction1DLattice(lattice_parameters)`
- `InterferenceFunction1DParaCrystal(peak_distance, width, corr_length)`
- `InterferenceFunction2DLattice(lattice_parameters)`

- `InterferenceFunction2DParaCrystal(length_1, length_2, α _lattice, ξ , corr_length)`

We are now going to describe these interference functions.



Remark: `InterferenceFunction1DLattice` can only be used for particles which are infinitely long in one of scattering plane's directions like for example a rectangular grating.

►► InterferenceFunctionNone()

The particles are placed randomly in the dilute limit and are considered as individual, non-interacting scatterers. The scattered intensity is function of the form factors only.

Example The sample is made of a substrate on which are deposited half-spheres. Script C.1 details the commands necessary to reproduce the output shown in fig. C.1.

Listing C.1: Python script to simulate a sample made of half-spheres deposited on a substrate layer without any interference. The part specific to the interferences is marked in red italic font.

```
import numpy
import matplotlib
import pylab
from libBornAgainCore import *

def get_sample():
    """
    Build and return the sample representing particles with no
    interference
    """
    # defining materials
    m_ambience = MaterialManager.getHomogeneousMaterial("Air",
        0.0, 0.0)
    m_substrate = MaterialManager.getHomogeneousMaterial("
        Substrate", 6e-6, 2e-8)
    m_particle = MaterialManager.getHomogeneousMaterial("Particle
        ", 6e-4, 2e-8)
    # collection of particles
    sphere_ff = FormFactorTruncatedSphere(5*nanometer, 5*
        nanometer)
    sphere = Particle(m_particle, sphere_ff)
    particle_layout = ParticleLayout()
    particle_layout.addParticle(sphere, 0.0, 1.0)
    interference = InterferenceFunctionNone()
    particle_layout.addInterferenceFunction(interference)

    air_layer = Layer(m_ambience)
    air_layer.setLayout(particle_layout)
    substrate_layer = Layer(m_substrate, 0)

    multi_layer = MultiLayer()
    multi_layer.addLayer(air_layer)
    multi_layer.addLayer(substrate_layer)
    return multi_layer

def get_simulation():
    """
    Create and return GISAXS simulation with beam and detector
```



```

"""
simulation = Simulation()
simulation.setDetectorParameters(100, 0.0*degree, 2.0*degree,
    100, 0.0*degree, 2.0*degree, True)
simulation.setBeamParameters(1.0*angstrom, 0.2*degree, 0.0*
    degree)
return simulation

def run_simulation():
    """
    Run simulation and plot results
    """
    sample = get_sample()
    simulation = get_simulation()
    simulation.setSample(sample)
    simulation.runSimulation()
    result = simulation.getIntensityData().getArray() + 1 # for
        log scale
    pylab.imshow(numpy.rot90(result, 1), norm=matplotlib.colors.
        LogNorm(), extent=[0.0, 2.0, 0, 2.0])
    pylab.show()

if __name__ == '__main__':
    run_simulation()

```

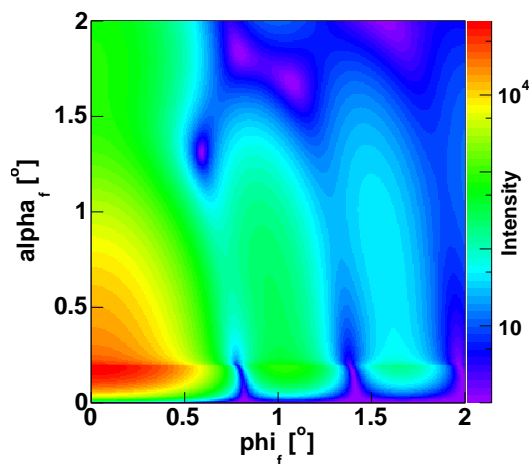


Figure C.1: Output intensity scattered from a sample made of half-spheres with **no interference** between them.

► `InterferenceFunction1DLattice(lattice_parameters)`

where `lattice_parameters=(lattice_length, ξ)` with `lattice_length` is the lattice constant and ξ the angle in radian between the lattice unit vector and the x -axis of the "GISAS experiment" referential as shown in fig. C.2.

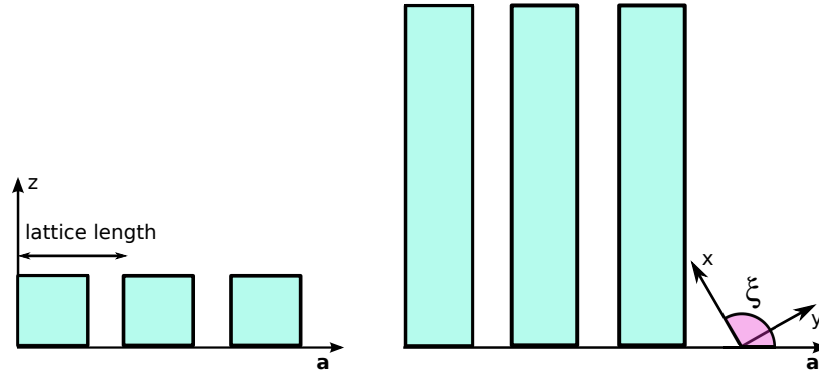


Figure C.2: Schematic representation of a 1D lattice (side and top views). Such a lattice is characterized by a lattice length and the angle ξ .



Remark: By default the long axis of the particles in this 1D lattice is along the beam axis: $\xi = 90^\circ$.

A probability distribution function pdf has to be chosen from the list in section C.2.2 in order to apply some modifications to the scattering peaks. This function is implemented using `setProbabilityDistributions(pdf)`.

Example Instead of giving a full script, whose output is the scattered intensity, we are focusing on how to build a sample using `InterferenceFunction1DLattice` as the interference function in BornAgainScript C.2 details this procedure in Python. As mentioned previously, this interference function can only be used with infinitely wide or long particles. Here the sample is made of infinitely long boxes deposited on a substrate (these particles are characterized by their widths and heights). They are also rotated by 90° in the scattering plane in order to have their long axis perpendicular to the input beam, which is along the x -axis. The lattice parameters (the lattice lengths and angle between the lattice main axis and the x -axis) are specified using `Lattice1DIFParameters()` and are then used as input parameters of the interference function.

Listing C.2: Python script to generate a sample made of half-spheres deposited on a substrate layer with the `1DLatticeInterference` function. The part specific to the interferences is marked in red italic font.

```
def get_sample():
    """
```

```

Build and return the sample with 1DLatticeInterference
function .
"""
# defining materials
m_air = MaterialManager.getHomogeneousMaterial("Air", 0.0,
0.0)
m_substrate = MaterialManager.getHomogeneousMaterial("
Substrate", 6e-6, 2e-8)
m_particle = MaterialManager.getHomogeneousMaterial("Particle
", 6e-4, 2e-8)

# collection of particles
ff = FormFactorInfLongBox(10.*nanometer, 15.0*nanometer)
box = Particle(m_particle, ff)
particle_layout = ParticleLayout()
transform = Transform3D.createRotateZ(90.0*degree)

particle_layout.addParticle(box, transform)

# lattice parameters
lattice_params = Lattice1DIFParameters()
lattice_params.m_length = 30.0*nanometer
lattice_params.m_xi = 0.0*degree

# interference function
interference = InterferenceFunction1DLattice(lattice_params)
pdf = FTDistribution1DCauchy(200./2./M_PI*nanometer)
interference.setProbabilityDistribution(pdf)
particle_decoration.addInterferenceFunction(interference)

# air layer with particles and substrate form multi layer
air_layer = Layer(m_air)
air_layer.setDecoration(particle_decoration)
substrate_layer = Layer(m_substrate, 0)

multi_layer = MultiLayer()
multi_layer.addLayer(air_layer)
multi_layer.addLayer(substrate_layer)
return multi_layer

```

► `InterferenceFunction1DParaCrystal(peak_distance, width, corr_length)`

where `peak_distance` is the average distance to the first neighbor peak,

`width` is the width parameter of the probability distribution,

`corr_length` is the correlation length (equal to 0 by default).

For this particular interference function, the implemented probability distribution function is Gaussian:

$$p(x) = \frac{1}{\omega\sqrt{2\pi}} \exp\left(-\frac{(x-D)^2}{\omega^2}\right), \quad P(q_{\parallel}) = \exp\left(-\frac{q_{\parallel}^2 \omega^2}{2}\right) \exp(i q_{\parallel} D)$$

where $\omega \equiv \text{width}$, $D \equiv \text{peak_distance}$, and $q_{\parallel} = \sqrt{\text{Re}^2(q_x) + \text{Re}^2(q_y)}$ (see fig. C.3).

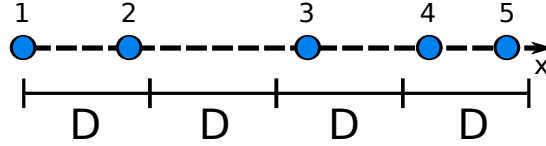


Figure C.3: Schematic representation of a 1D paracrystal in real space (side view). D is the average spacing between the particles.

$$S_{\text{1DParaCrystal}}(q_{\parallel}) = \text{Re} \left(\frac{1 + \Phi(q_{\parallel})}{1 - \Phi(q_{\parallel})} \right),$$

$$\text{where } \Phi(q_{\parallel}) = \begin{cases} P(q_{\parallel}) & \text{if } \text{corr_length} = 0 \\ P(q_{\parallel}) \exp\left(-\frac{D}{\text{corr_length}}\right) & \text{otherwise} \end{cases}$$

Example To illustrate the 1D paracrystal interference function, we use the same sample as in the case without interference: half-spheres deposited on a substrate.

Listing C.3: Python script to simulate a sample made of half-spheres deposited on a substrate layer and interfering with the "1D paracrystal" model. The part specific to the interferences is marked in red italic font.

```
import numpy
import matplotlib
import pylab
from libBornAgainCore import *

def get_sample():
    """
    Build and return the sample representing particles with 1D
    paracrystal
```

```

"""
# defining materials
m_ambience = MaterialManager.getHomogeneousMaterial("Air",
    0.0, 0.0)
m_substrate = MaterialManager.getHomogeneousMaterial("
    Substrate", 6e-6, 2e-8)
m_particle = MaterialManager.getHomogeneousMaterial("Particle
    ", 6e-4, 2e-8)
# collection of particles
sphere_ff = FormFactorTruncatedSphere(5*nanometer, 5*
    nanometer)
sphere = Particle(m_particle, sphere_ff)
particle_layout = ParticleLayout()
particle_layout.addParticle(sphere, 0.0, 1.0)
interference = InterferenceFunction1DParaCrystal(25.0*
    nanometer, 7*nanometer, 1e3*nanometer)
particle_layout.addInterferenceFunction(interference)

air_layer = Layer(m_ambience)
air_layer.setLayout(particle_layout)
substrate_layer = Layer(m_substrate, 0)

multi_layer = MultiLayer()
multi_layer.addLayer(air_layer)
multi_layer.addLayer(substrate_layer)
return multi_layer

def get_simulation():
    """
    Create and return GISAXS simulation with beam and detector
    """
    simulation = Simulation()
    simulation.setDetectorParameters(100, 0.0*degree, 2.0*degree,
        100, 0.0*degree, 2.0*degree, True)
    simulation.setBeamParameters(1.0*angstrom, 0.2*degree, 0.0*
        degree)
    return simulation

def run_simulation():
    """
    Run simulation and plot results
    """
    sample = get_sample()
    simulation = get_simulation()
    simulation.setSample(sample)
    simulation.runSimulation()
    result = simulation.getIntensityData().getArray() + 1 # for
        log scale
    pylab.imshow(numpy.rot90(result, 1), norm=matplotlib.colors.

```

```
LogNorm(), extent=[0.0, 2.0, 0, 2.0])
pylab.show()

if __name__ == '__main__':
    run_simulation()
```

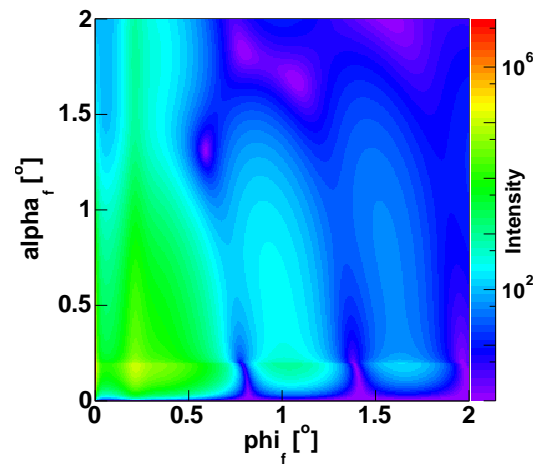


Figure C.4: Output intensity scattered from a sample made of half-spheres with "1Dparacrystal" interference between them. This figure has been generated using Script C.3.

► `InterferenceFunction2DLattice(lattice_parameters)`

where `lattice_parameters` corresponds to (L_1, L_2, α, ξ) (see illustration in figure C.6) with

L_1, L_2 the lengths of the lattice cell,

α the angle between the lattice basis vectors \mathbf{a}, \mathbf{b} in direct space,

ξ is the angle defining the lattice orientation (set to 0 by default); it is taken as the angle between the \mathbf{a} vector of the lattice basis and the \mathbf{x} axis of the "GISAS experiment" referential (as shown in figure 3.1).

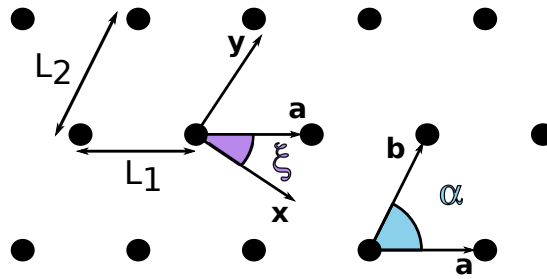


Figure C.5: Schematic representation of a 2D lattice (top view). Such a lattice is characterized by lattice lengths L_1, L_2 and angles α and ξ .

Like for the one-dimensional case, a probability distribution function pdf has to be defined. One can choose between those listed in section C.2.2. This function is implemented using `setProbabilityDistributions(pdf)`.

Example The sample used to run the simulation is made of half-spheres deposited on a substrate. The interference function is "2DLattice" and the particles are located at the nodes of a square lattice with $L_1 = L_2 = 20$ nm, $\mathbf{a} \equiv \mathbf{b}$ and the probability distribution function is Gaussian. We also use the Local Monodisperse Approximation.

Listing C.4: Python script to simulate a sample made of half-spheres deposited on a substrate layer with "2DLattice" interference function. The part specific to the interferences is marked in red italic font.

```
import numpy
import matplotlib
import pylab
from libBornAgainCore import *

def get_sample():
    """
    Build and return the sample representing particles with 2D
    lattice interference
```

```

"""
# defining materials
m_ambience = MaterialManager.getHomogeneousMaterial("Air",
    0.0, 0.0)
m_substrate = MaterialManager.getHomogeneousMaterial("
    Substrate", 6e-6, 2e-8)
m_particle = MaterialManager.getHomogeneousMaterial("Particle
    ", 6e-4, 2e-8)

# lattice parameters
lattice_params = Lattice2DIFParameters()
lattice_params.m_length_1 = 20.0*nanometer
lattice_params.m_length_2 = 20.0*nanometer
lattice_params.m_angle = 90.0*degree
lattice_params.m_xi = 0.0*degree

#collection of particles
sphere_ff = FormFactorTruncatedSphere(5*nanometer, 5*
    nanometer)
sphere = Particle(m_particle, sphere_ff)

interference = InterferenceFunction2DLattice(lattice_params)
pdf = FTDistribution2DGauss(200.0*nanometer/2.0/M_PI, 75.0*
    nanometer/2.0/M_PI)
interference.setProbabilityDistribution(pdf)
particle_layout = ParticleLayout()
particle_layout.addParticle(sphere, 0.0, 1.0)
particle_layout.addInterferenceFunction(interference)

air_layer = Layer(m_ambience)
air_layer.setLayout(particle_layout)
substrate_layer = Layer(m_substrate, 0)
multi_layer = MultiLayer()
multi_layer.addLayer(air_layer)
multi_layer.addLayer(substrate_layer)
return multi_layer

def get_simulation():
    """
    Create and return GISAXS simulation with beam and detector
    """
    simulation = Simulation()
    simulation.setDetectorParameters(100, 0.0*degree, 2.0*degree,
        100, 0.0*degree, 2.0*degree, True)
    simulation.setBeamParameters(1.0*angstrom, 0.2*degree, 0.0*
        degree)
    sim_params = SimulationParameters()
    sim_params.me_if_approx = SimulationParameters.LMA

```



```

    simulation.setSimulationParameters(sim_params)
    return simulation

def run_simulation():
    """
    Run simulation and plot results
    """
    sample = get_sample()
    simulation = get_simulation()
    simulation.setSample(sample)
    simulation.runSimulation()
    result = simulation.getIntensityData().getArray() + 1 # for
        log scale
    pylab.imshow(numpy.rot90(result, 1), norm=matplotlib.colors.
        LogNorm(), extent=[0.0, 2.0, 0, 2.0])
    pylab.show()

if __name__ == '__main__':
    run_simulation()

```

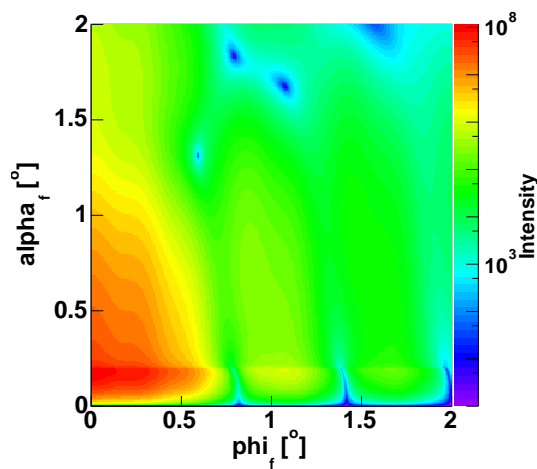


Figure C.6: Output intensity scattered from a sample made of half-spheres with **2DLattice** interference function.

► `InterferenceFunction2DParaCrystal(L_1, L_2, lattice_angle, ξ , corr_length)`

where L_1, L_2 are the lengths of the lattice cell,

`lattice_angle` the angle between the lattice basis vectors **a**, **b** in direct space,

ξ is the angle defining the lattice orientation (set to 0 by default).

Two special configurations have also been implemented:

- `createSquare(peak_distance, corr_length, domain_size_1, domain_size_2)`
where the angle between the base vectors of the lattice is set to $\pi/2$,
- `createHexagonal(peak_distance, corr_length, domain_size_1, domain_size_2)`
where the angle between the base vectors of the lattice is set to $2\pi/3$,

where `domain_size1, 2` are the dimensions of the paracrystal along the main axes,

`peak_distance` is the same in both directions and **a** \equiv **x**.

Probability distribution functions have to be defined. As the two-dimensional paracrystal is defined from two independent 1D paracrystals, we need two of these functions, using `setProbabilityDistributions(pdf_1, pdf_2)`, with `pdf_1, 2` are related to each main axis of the paracrystal.

Example The particles deposited on a substrate are half-spheres. They interference via the 2DParacrystal distribution function. The paracrystal is based on a 2D hexagonal lattice with a Gaussian probability distribution function in reciprocal space.

Listing C.5: Python script to simulate a sample made of half-spheres deposited on a substrate layer with "2DParacrystal" interference function. The part specific to the interferences is marked in red italic font.

```
import numpy
import matplotlib
import pylab
from libBornAgainCore import *

def get_sample():
    """
    Build and return the sample representing 2D paracrystal
    """
    m_ambience = MaterialManager.getHomogeneousMaterial("Air",
        0.0, 0.0)
    m_substrate = MaterialManager.getHomogeneousMaterial("
        Substrate", 6e-6, 2e-8)
    m_particle = MaterialManager.getHomogeneousMaterial("Particle
        ", 6e-4, 2e-8)

    # collection of particles
    sphere_ff = FormFactorTruncatedSphere(5*nanometer, 5*
        nanometer)
```

```

sphere = Particle(m_particle, sphere_ff)
particle_decoration = ParticleDecoration()
particle_decoration.addParticle(sphere, 0.0, 1.0)

interference = InterferenceFunction2DParaCrystal.
    createHexagonal(30.0*nanometer, 0.0, 40.0*micrometer,
    40.0*micrometer)
pdf = FTDistribution2DCauchy(1.0*nanometer, 1.0*nanometer)
interference.setProbabilityDistributions(pdf, pdf)
particle_decoration.addInterferenceFunction(interference)

air_layer = Layer(m_ambience)
air_layer.setDecoration(particle_decoration)

substrate_layer = Layer(m_substrate, 0)

multi_layer = MultiLayer()
multi_layer.addLayer(air_layer)
multi_layer.addLayer(substrate_layer)
return multi_layer

def get_simulation():
    """
    Create and return GISAXS simulation with beam and detector
    defined
    """
    simulation = Simulation()
    simulation.setDetectorParameters(100, 0.0*degree, 2.0*degree,
    100, 0.0*degree, 2.0*degree, True)
    simulation.setBeamParameters(1.*angstrom, 0.2*degree, 0.0*
    degree)
    return simulation

def run_simulation():
    """
    Run simulation and plot results
    """
    sample = get_sample()
    simulation = get_simulation()
    simulation.setSample(sample)
    simulation.runSimulation()
    result = simulation.getIntensityData().getArray() + 1
    pylab.imshow(numpy.rot90(result, 1), norm=matplotlib.colors.
    LogNorm(), extent=[0.0, 2.0, 0, 2.0])
    pylab.show()

if __name__ == '__main__':

```

```
run_simulation()
```

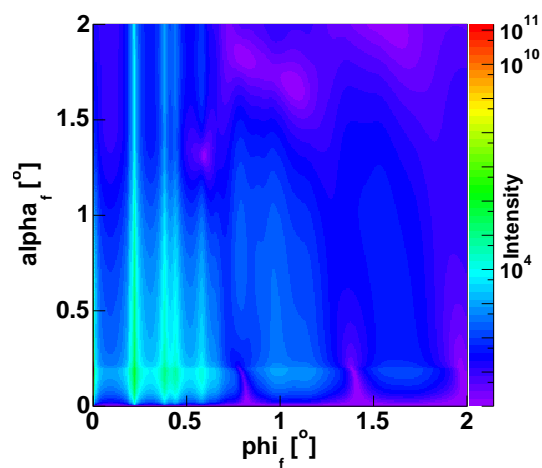


Figure C.7: Output intensity scattered from a sample made of half-spheres with **2DParacrystal** interference function.

C.3 Summary

Function	Parameters	Comments
InterferenceFunctionNone C.2.3	None	disordered distribution
InterferenceFunction1DLattice C.2.3	lattice_length $\xi = \widehat{(\mathbf{x}, \mathbf{a})}$	use only with infinitely long/wide particles pdf=(Cauchy, Gauss or Voigt) to be defined
InterferenceFunction1DParaCrystal C.2.3	peak_distance of pdf width of pdf corr_length (optional)	only Gaussian pdf implemented (no option)
InterferenceFunction2DLattice C.2.3	L_1, L_2: lattice lengths lattice_angle= $\widehat{(\mathbf{a}, \mathbf{b})}$ $\xi = \widehat{(\mathbf{x}, \mathbf{a})}$	pdf=(Cauchy, Gauss or Voigt) to be defined
InterferenceFunction2DParaCrystal C.2.3	L_1, L_2: lattice lengths lattice_angle= $\widehat{(\mathbf{a}, \mathbf{b})}$ $\xi = \widehat{(\mathbf{x}, \mathbf{a})}$ corr_length (optional) same for both axes	2D pdf=(Cauchy, Gauss or Voigt) to be defined (1 pdf per axis)

Table C.2: List of interference functions implemented in BornAgain. pdf : probability distribution function, \mathbf{a}, \mathbf{b} are the lattice base vector, and \mathbf{x} is the axis vector perpendicular to the detector plane.

Bibliography

- [1] R. Lazzari, J. Appl. Cryst. **35**, 406–421 (2002).
- [2] M. Lutz, *Python pocket reference*, O'Reilly media (⁴2009).
- [3] <http://www.numpy.org>.
- [4] Minuit user's guide, <http://seal.web.cern.ch/seal/documents/minuit/mnusersguide.pdf>.
- [5] <http://seal.web.cern.ch/seal/documents/minuit/mntutorial.pdf>.
- [6] http://www.gnu.org/software/gsl/manual/html_node/Multidimensional-Minimization.html.
- [7] http://www.gnu.org/software/gsl/manual/html_node/Nonlinear-Least_002dSquares-Fitting.html#Nonlinear-Least_002dSquares-Fitting.
- [8] http://www.gnu.org/software/gsl/manual/html_node/Simulated-Annealing.html.
- [9] M. Abramowitz and I. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards (1964).
- [10] P. L. G, Phys. Rev. E **95**, 359 (1954).
- [11] M. Born and E. Wolf, *Principles of Optics*, Cambridge University Press (1999).
- [12] G. Renaud, R. Lazzari and F. Leroy, Surface Science Reports **64**, 255 (2009).
- [13] J. K. Pedersen, J. Appl. Cryst. **27**, 595 (1994).
- [14] F. Leroy, R. Lazzari and G. Renaud, Acta Cryst. A **60**, 565 (2004).
- [15] R. Hosemann, Acta Cryst. **4**, 520 (1951).