BornAgain

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

User Manual
0.2.1
November 21, 2013

C. Durniak, 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

`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, 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 0.2.1,
> `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: install 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 (≥ 3.3.1)

- `Python`-2.7, python-devel, python-numpy-devel

Step II: get 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: build 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: run an example

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

1.2   Quick start 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 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: use 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: run the example
Run an example located in BornAgain installation directory:

```
python C:/BornAgain-0.9.2/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)

- 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 $\geq$ 3.1 or GCC versions $\geq$ 4.2

- cmake ($\geq$ 2.8.3)

- boost library ($\geq$ 1.48)

- GNU scientific library ($\geq$ 1.15)

- fftw3 library ($\geq$ 3.3)

- `Python` ($\geq$ 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

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 xfz 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-0.9.2/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.
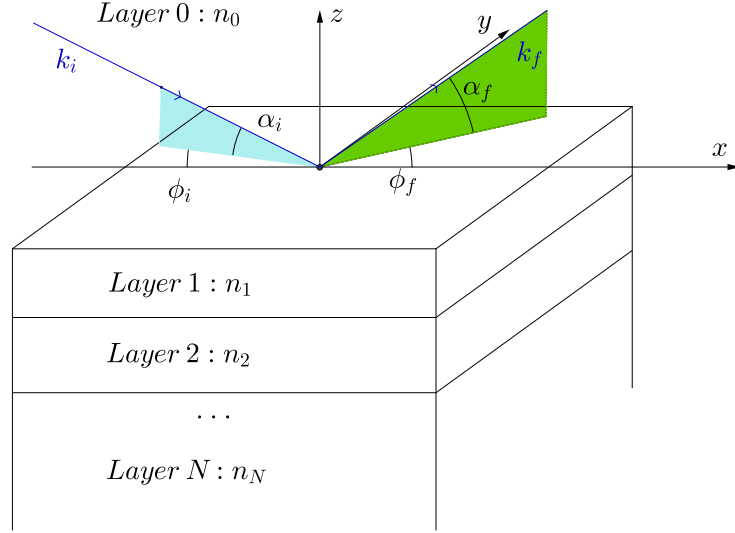

Figure 3.1: Representation of the scattering geometry. n_j is the refractive index of layer j and α_i and ϕ_i are the incident angles of the wave propagating. α_f is the exit angle with respect to the sample's surface and ϕ_f is the scattering angle with respect to the scattering plane.

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 the 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.

3.2.1 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
           pyramids on top of
8      substrate without interference.
9      """
10     # defining materials
11     m_air = MaterialManager.getHomogeneousMaterial("Air", 0.0,
           0.0)
12     m_substrate = MaterialManager.getHomogeneousMaterial("
           Substrate", 6e-6, 2e-8)
13     m_particle = MaterialManager.getHomogeneousMaterial("Particle
           ", 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(5*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, 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 `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 `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).

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()
```

In function defined starting from the line 41 we bring all items together. 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_phi×n_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.
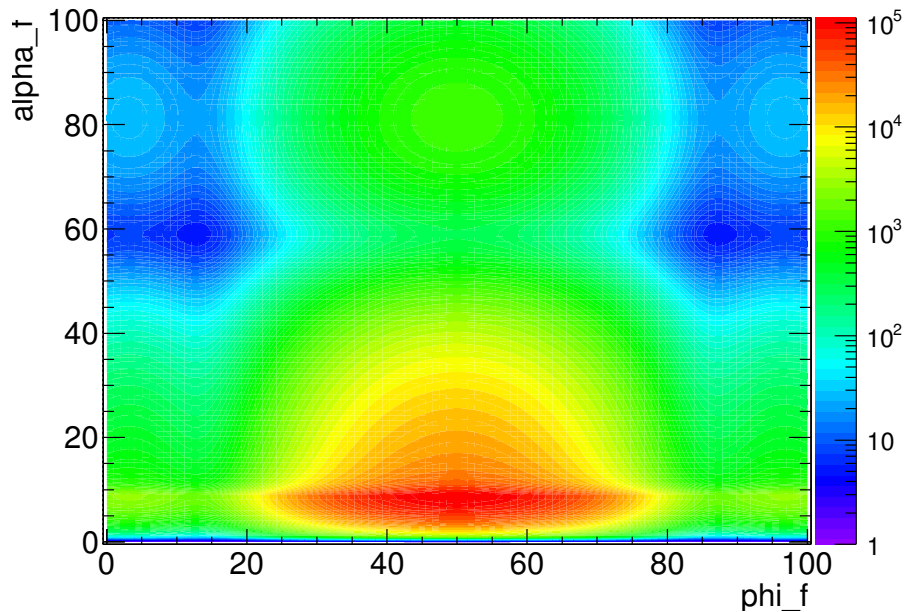
Figure 3.2: Simulated grazing-incidence small-angle X-ray scattering from a mixture of cylindrical and prismatic nanoparticles without any interference, deposited on top of a substrate. The input beam is characterized by a wavelength λ of 1 Å and incident angles $\alpha_i = 0.2°$, $\phi_i = 0°$. The cylinders have a radius and a height both equal to 5 nm, the prisms are characterized by a side length equal to 5 nm and they are also 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.

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

```
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/half_side':5
'/MultiLayer/Layer0/ParticleDecoration/ParticleInfo1/Particle/
    FormFactorPrism3/height':5
```
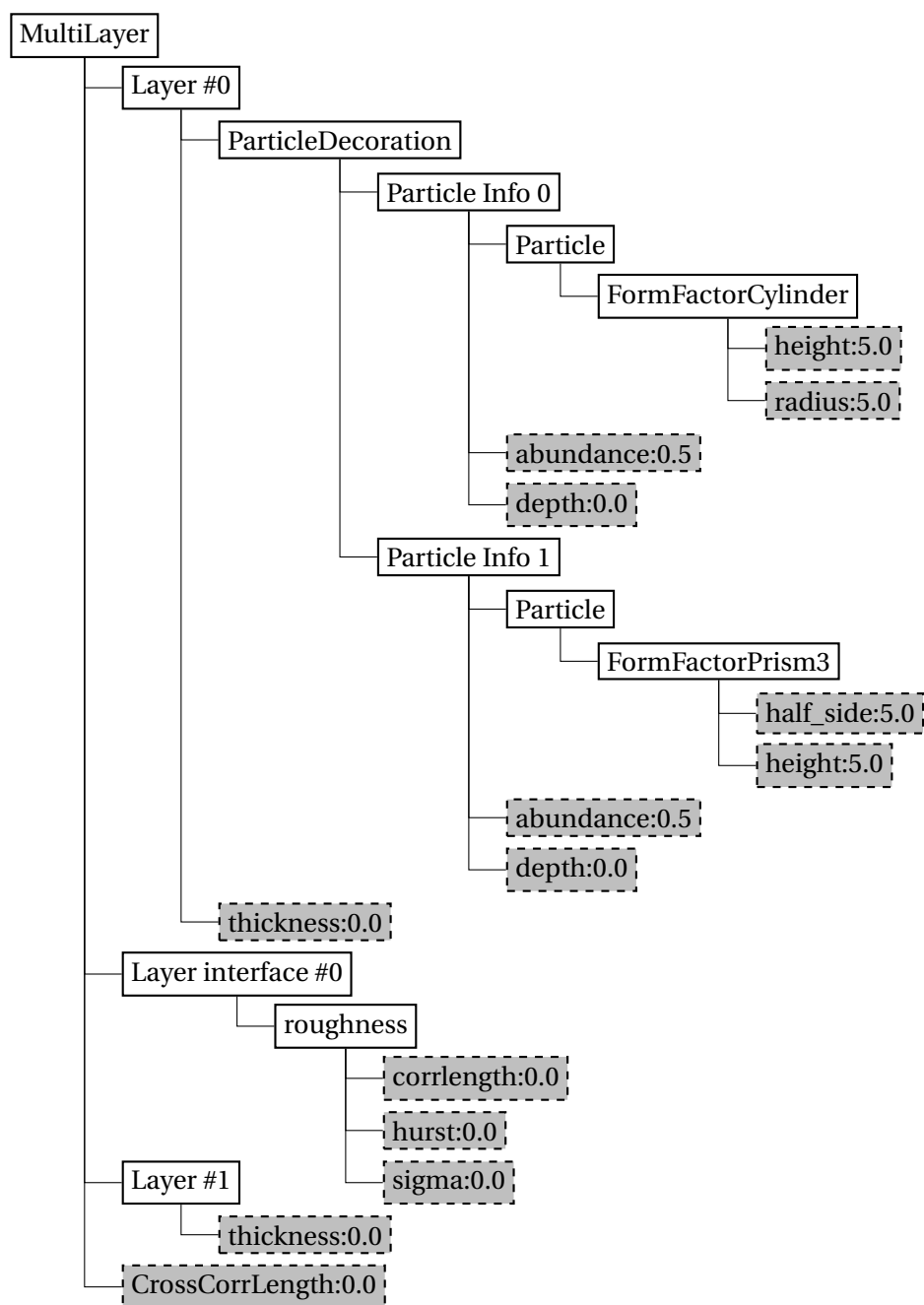
Figure 3.3: Tree representation of the sample structure.

```
'/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 multi-layered 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 fit 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 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).
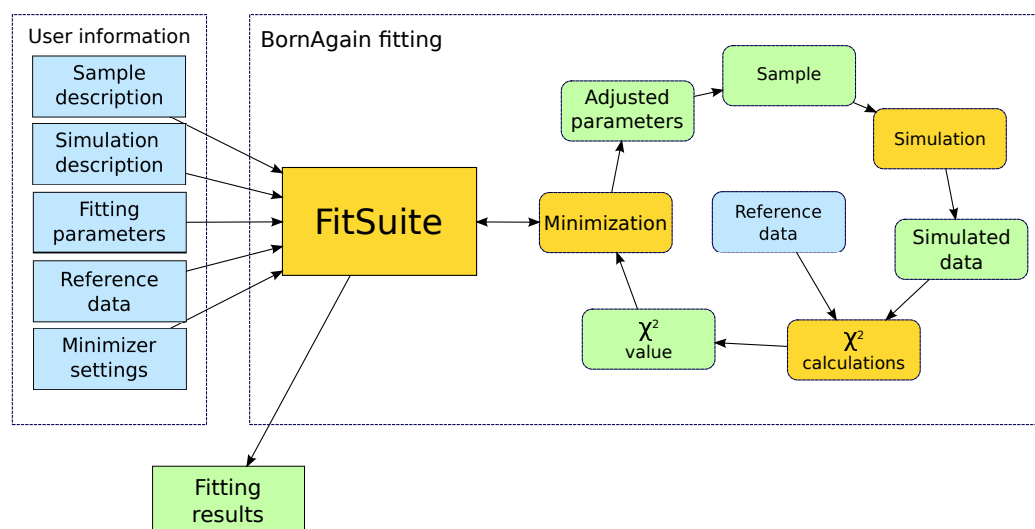
Figure 4.1: Fitting work flow.

- 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 $\chi2$ 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 `<chi2_module>` parameter, different normalizations and weights can be applied to give user full control of the way $\chi2$ 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
`GSLMultiFit` [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:

```
options = MinimizerOptions()
options.setMaxFunctionCalls(10)
fit_suite.getMinimizer().setOptions(options)
```

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 ant 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 unkown

- the radius of cylinders,

- the height of cylinders,

- the half side 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 all four of these parameters. 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 length 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
          of substrate without interference.
8      """
9      # defining materials
10     m_air = MaterialManager.getHomogeneousMaterial("Air", 0.0,
          0.0)
11     m_substrate = MaterialManager.getHomogeneousMaterial("
          Substrate", 6e-6, 2e-8)
12     m_particle = MaterialManager.getHomogeneousMaterial("Particle
          ", 6e-4, 2e-8)
13
14     # collection of particles
15     cylinder_ff = FormFactorCylinder(1.0*nanometer, 1.0*nanometer
          )
16     cylinder = Particle(m_particle, cylinder_ff)
17     prism_ff = FormFactorPrism3(1.0*nanometer, 1.0*nanometer)
18     prism = Particle(m_particle, prism_ff)
19     particle_decoration = ParticleDecoration()
20     particle_decoration.addParticle(cylinder, 0.0, 0.5)
21     particle_decoration.addParticle(prism, 0.0, 0.5)
22     interference = InterferenceFunctionNone()
23     particle_decoration.addInterferenceFunction(interference)
24
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)
29     multi_layer = MultiLayer()
30     multi_layer.addLayer(air_layer)
31     multi_layer.addLayer(substrate_layer)
```

```
32        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
            , 100, 0.0* degree, 2.0* degree, True)
41      simulation.setBeamParameters (1.0* angstrom, 0.2* degree, 0.0*
            degree)
42      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 ('
            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/half_side", 6.*
            nanometer, 0.01*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 half side length. The cylinder's length and prism half side are initially equal to 4 nm, whereas the cylinder's radius and the prism 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 $\chi2$ 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 unappropriate 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
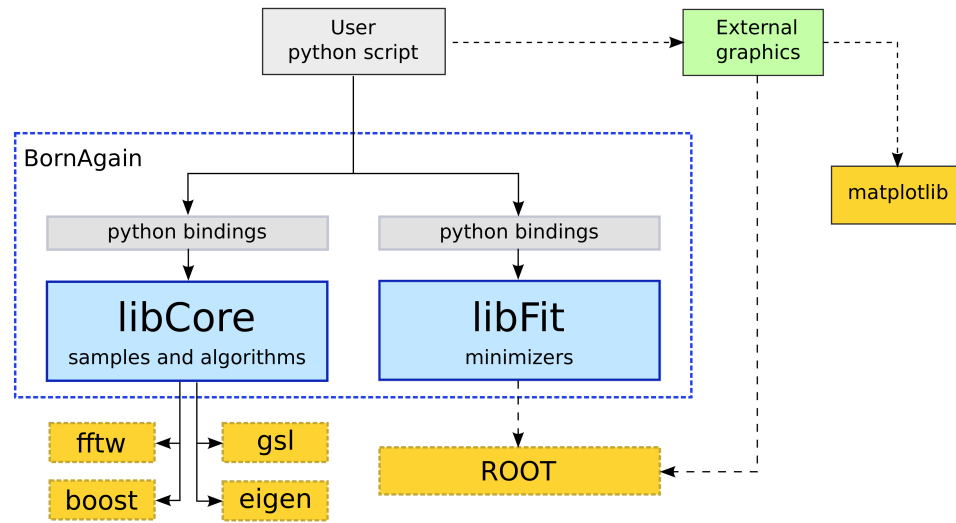
Figure 5.1: Structure of `BornAgain` libraries.

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.
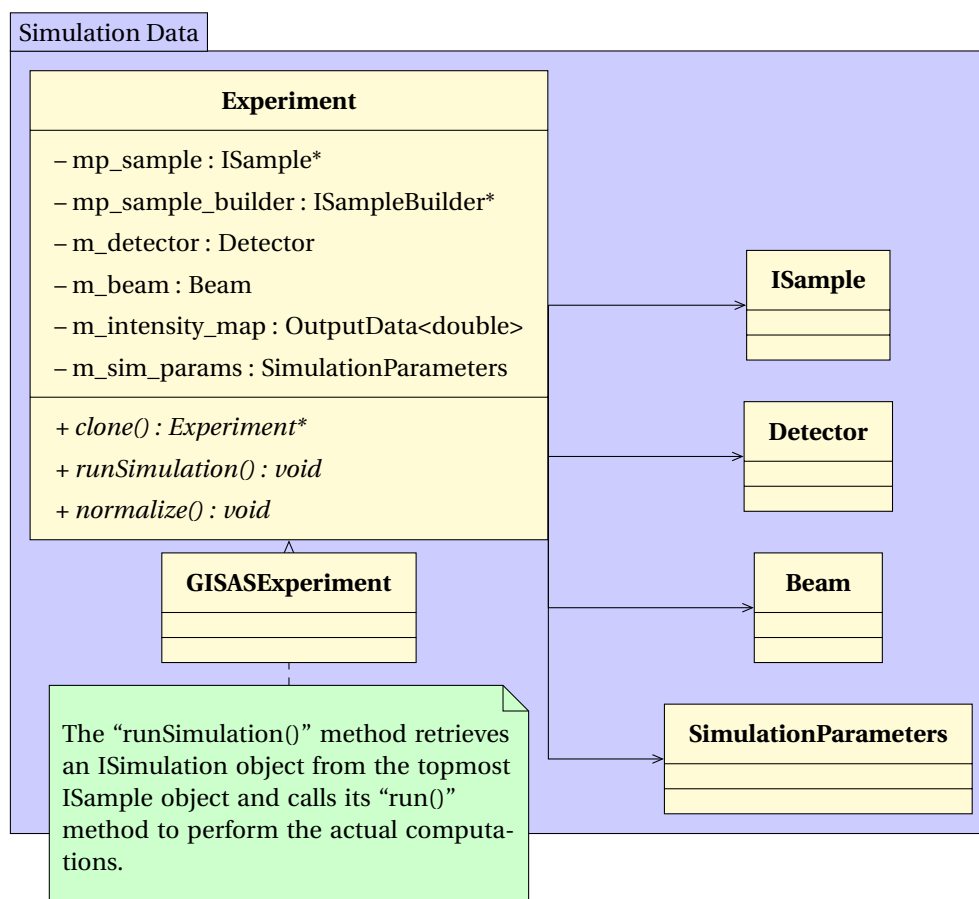

Figure 5.2: The Experiment class as a container for sample, beam, detector and simulation parameters.

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.

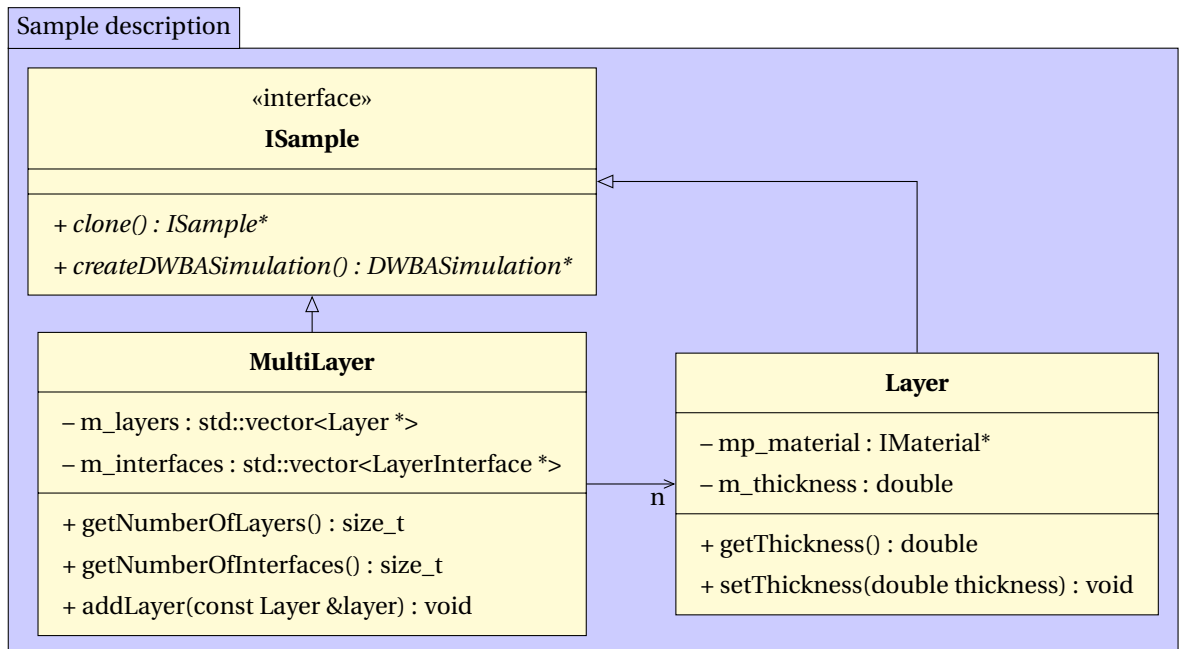This interface and two of its subclasses are sketched in figure 5.3.

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
```

```python
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
           pyramids on top of
10     substrate without interference.
11     """
12     # defining materials
13     m_air = MaterialManager.getHomogeneousMaterial("Air", 0.0,
           0.0)
14     m_substrate = MaterialManager.getHomogeneousMaterial("
           Substrate", 6e-6, 2e-8)
15     m_particle = MaterialManager.getHomogeneousMaterial("Particle
           ", 6e-4, 2e-8)
16
17     # collection of particles
18     cylinder_ff = FormFactorCylinder(5*nanometer, 5*nanometer)
19     cylinder = Particle(m_particle, cylinder_ff)
20     prism_ff = FormFactorPrism3(5*nanometer, 5*nanometer)
21     prism = Particle(m_particle, prism_ff)
22     particle_decoration = ParticleDecoration()
23     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
            defined
41        """
42        simulation = Simulation()
43        simulation.setDetectorParameters(100, -1.0*degree, 1.0*degree
            , 100, 0.0*degree, 2.0*degree, True)
44        simulation.setBeamParameters(1.0*angstrom, 0.2*degree, 0.0*
            degree)
45        return simulation
46
47
48  def run_simulation():
49        """
50        Run simulation and plot results
51        """
52        sample = get_sample()
53        simulation = get_simulation()
54        simulation.setSample(sample)
55        simulation.runSimulation()
56        result = simulation.getIntensityData().getArray() + 1  # for
             log scale
57        pylab.imshow(numpy.rot90(result, 1), norm=matplotlib.colors.
            LogNorm(), extent=[-1.0, 1.0, 0, 2.0])
58        pylab.show()
59
60
61  if __name__ == '__main__':
62        run_simulation()
```

A.2   Python fitting example

The following script can be found at

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

```python
 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(1.0*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      """
```

```python
39      simulation = Simulation()
40      simulation.setDetectorParameters(100, -1.0*degree, 1.0*degree
            , 100, 0.0*degree, 2.0*degree, True)
41      simulation.setBeamParameters(1.0*angstrom, 0.2*degree, 0.0*
            degree)
42      return simulation
43
44
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('
            refdata_fitcylinderprisms.txt')
54
55      fit_suite = FitSuite()
56      fit_suite.addSimulationAndRealData(simulation, real_data)
57      fit_suite.initPrint(10)
58
59      # setting fitting parameters with starting values
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/half_side", 6.*
            nanometer, 0.01*nanometer, AttLimits.lowerLimited(0.01))
64
65      # running fit
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()
74
75  if __name__ == '__main__':
76      run_fitting()
```
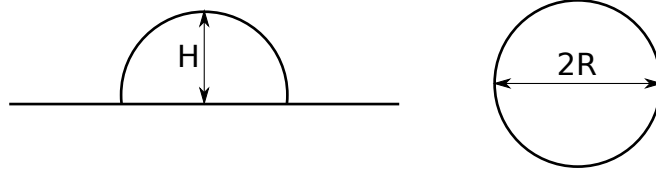
Appendix B

Formfactors

- Parallelepiped
- Pyramid
- Cylinder
- Cone
- Prism3
- Tethraedron
- Prism6
- Cone6
- Cut-off sphere, Section B.1
- Cybooctaedron
- Facetted sphere
- Full sphere, Section B.2
- Full spheroid
- Box
- Anisotropic pyramid
- Ellipsoid
- Anisotropic hemi-spheroid
- Spheroid

B.1 Formfactor Cut-off Sphere

B.1.1 Real-space geometry

A sphere, with a planar cut-off.

Parameters:

- radius R

- heigth H

Restrictions:

- $H \leq 2R$

Properties:

- volume $V = \pi R^3 \left[\dfrac{2}{3} + \dfrac{H-R}{R} - \dfrac{1}{3} \left(\dfrac{H-R}{R} \right)^3 \right]$

- particle surface seen from above $S = \begin{cases} \pi R^2, & H > R \\ \pi \left(2RH - H^2 \right), & H < R \end{cases}$.

- gyration radius along z axis $R_g = \begin{cases} R, & H > R \\ \sqrt{2RH - H^2}, & H < R \end{cases}$.

B.1.2 Computing the formfactor

The formfactor can be computed analytically upto a 1-dimensional quadrature:

$$F(\mathbf{q}, R, H) = \exp\left[i q_z (H - R) \right] \int\limits_{R-H}^{R} 2\pi R_z^2 \frac{J_1(q_{\parallel} R_z)}{q_{\parallel} R_z} \exp\left[i q_z z \right] dz \tag{B.1}$$

with abbreviations

$$q_{\parallel} \;\; = \;\; \sqrt{q_x^2 + q_y^2} \tag{B.2}$$
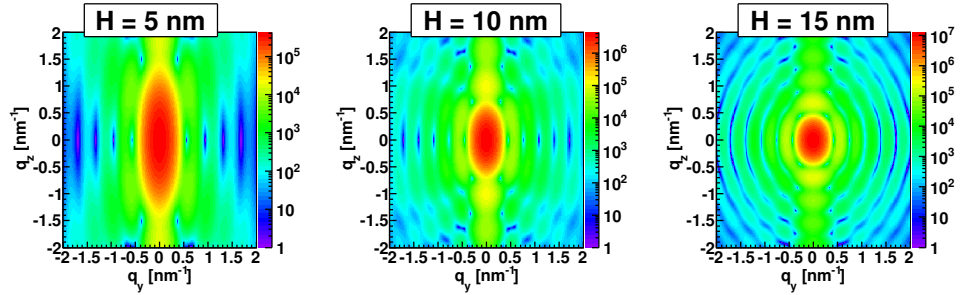
$$R_z \;\; = \;\; \sqrt{R^2 - z^2} \tag{B.3}$$

B.1.3 Exemplary formfactor

$|F|^2$, computed for $R = 10$ nm and $H = 13$ nm.
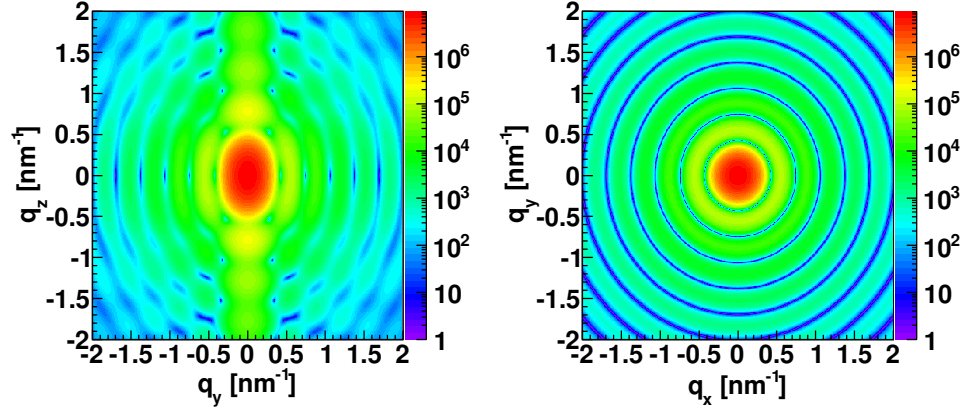
B.1.4 Parameter dependence

$|F|^2$, computed for $R = 10$ nm and $H = 5$, 10 and 15 nm.

B.1.5 Related particle shapes

Full sphere (Section B.2), Ellipsoid, Full spheroid, Hemi-ellipsoid.

B.1.6 References

Equation (B.1) agrees with the "sphere" formfactor of `IsGISAXS` [1].

B.2 Formfactor Full Sphere

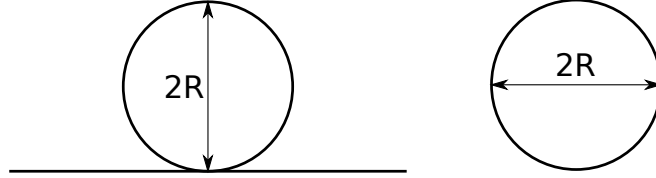Formfactor full sphere is represented as a sphere with radius R.

Figure B.1: Sketch of the formfactor full sphere. Left: front view, right: top view.

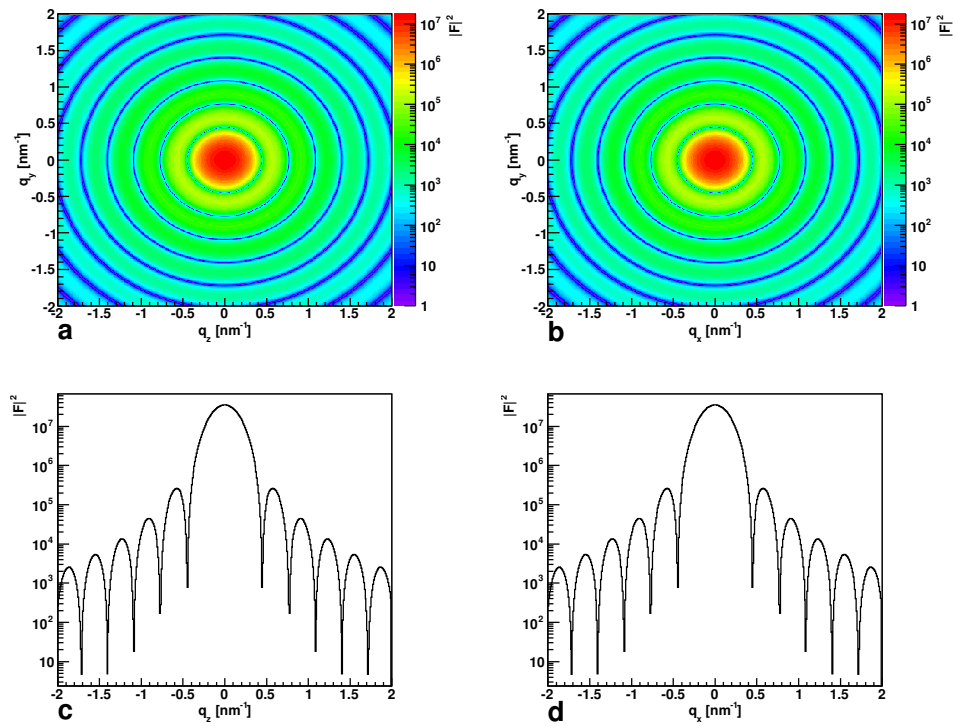$$F(\mathbf{q}, R) = 4\pi R \times \frac{\sin(qR) - qR\cos(qR)}{(qR)^3} \times \exp\left(i\, q_z R\right) \tag{B.4}$$

Figure B.2: **a:** $|F|^2$ plotted against q_z and q_y, **b:** $|F|^2$ plotted against q_x and q_y, **c:** slice of the picture **a** along the q_z axis, **d:** slice of the picture **b** along the q_x axis.

Bibliography

[1] R. Lazzari, J. Appl. Cryst. **35**, 406–421 (2002).

[2] M. Lutz, *Python pocket reference*, O'Reilly media (^{4}2009).

[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`.