BornAgain

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

User Manual
version 0.1.1
November 7, 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 change 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 [**?**].

However, BornAgain also extends this functionality by supporting an unrestricted number of layers and particles, diffuse reflection from rough layer interfaces, particles with inner structures and support for polarized neutrons 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 platform-independent software, with active support for Linux, MacOS and Microsoft Windows. It is a free and open source software provided under terms of 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 ⟨...⟩,
> `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 that is 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. More details about installation procedure are given in Section 2.

Step I: installing 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: getting the source
Download BornAgain source tarball from http://apps.jcns.fz-juelich.de/BornAgain or use git repository

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

Step III: building the source

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

Step IV: running example

```
cd <install_dir>/Examples/python/ex001_CylindersAndPrisms
python CylindersAndPrisms.py
```

1.2 Quick start on Windows Platforms

Step I: installing 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 at https://code.google.com/p/pythonxy which, with default installation options, will contain at least these three packages. BornAgain installation.

Step II: using installation package
The Windows installation package can be downloaded from http://apps.jcns.fz-juelich.de/BornAgain. Double click it to start installation process, then follow instructions.

Step III: running example
Run an example simulation by double-clicking on the python script located in the BornAgain installation directory:

```
python C:/BornAgain-0.9.1/Examples/python/
   ex001_CylindersAndPrisms/CylindersAndPrisms.py
```

1.3 Getting help

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

Chapter 2

Installation

BornAgain is intended to work on x86/x86_64 Linux, Mac OS X and Windows operating systems. It was 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 package on MS Windows 7,8 (see Section 2.1 and Section 2.2). In the next releases we are planning to provide binary installers for Mac OS X and Debian.

We welcome user feedback and/or bug reports related to they installation experience 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 building BornAgain :

1. Acquire required third-party libraries.
2. Get BornAgain source code.
3. Use CMake to build and install 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 ≥ 4.2
- cmake (≥ 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 a few examples for several selected operation systems. Please note, that other distributions (Fedora, Mint, etc) may have different commands for invoking the package manager and slightly different names of packages (like "boost" instead of "libboost" etc). Besides that, the installation should be very similar.

Ubuntu (12.10, 13.04), Debian (7.1)
Installing required packages

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

Installing 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 "scientific" repository

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

Installing required packages

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

Installing 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 terminal window of your Mac.

Installing 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
```

Installing 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 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 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 updating 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 to get the most recent features one can switch source code to it by

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

2.1.3 Building and installing the code

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

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

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

About CMake

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

Compilation process invoked by the command “make” lasts about 10 min for 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. Every 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 compiled libraries and some usage examples into the installation directory.

```
make install
```

Troubleshooting

In the case of complex system setup, with variety of libraries of different versions scattered across multiple places (`/opt/local`, `/usr` etc.), you may want to help `CMake` to find libraries in proper place. In example below two system variables are defined to force `CMake` to prefer libraries found in `/opt/local` to other places.

```
export CMAKE_LIBRARY_PATH=/opt/local/lib:$CMAKE_LIBRARY_PATH
export CMAKE_INCLUDE_PATH=/opt/local/include:$CMAKE_INCLUDE_PATH
```

2.1.4 Running first simulation

In your installation directory you will find

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

Run your first example and enjoy first BornAgain simulation plot.

```
cd <install_dir>/Examples/python/ex001_CylindersAndPrisms
python CylindersAndPrisms.py
```

2.2 Installing on Windows Platforms.

Step I: installing 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 at `https://code.google.com/p/pythonxy` which, with default installation options, will contain at least these three packages. The user has to download and install this package before proceeding with `BornAgain` installation.

Step II: using installation package
The Windows installation package can be downloaded from `http://apps.jcns.fz-juelich.de/BornAgain`. Double click it to start the installation process, then follow the instructions.

Step IV: running example
Run an example by double-clicking on the python script located in the `BornAgain` installation directory:

```
python C:/BornAgain-0.9.1/Examples/python/
   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 specifiying 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.

User defines all these steps using `BornAgain` API in Python script and then run the simulation by executing the script in Python interpreter. More information about general software architecture and `BornAgain` internal design are given in Section 5.

3.2 Conventions

3.2.1 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 plane of the detector and the y-axis is along it. 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

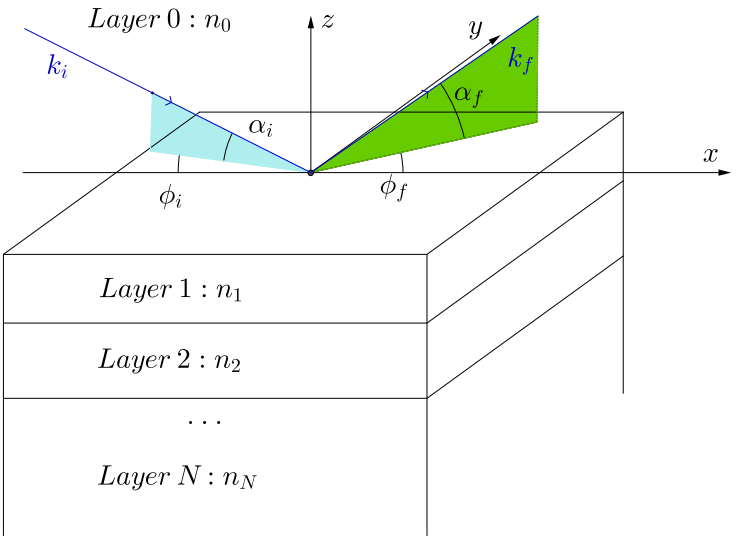

Figure 3.1: Representation of the scattering geometry. n_j is the refractive index of layer j and α_i and ϕ_i are the incident angle of the wave propagating. α_f is the exit angle with respect to the sample's surface and ϕ_f is the scattering angle with respect to the scattering plane.

for the angles α_i and α_f is so that they are positive as shown in figure 3.1.

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. We do not define any dimensions in the x, y directions. And, except for roughness, the layer's vertical boundaries 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.

Remark: - Order of the different steps for the simulation:
When assembling the sample, the layers are defined from top to bottom. So in most cases the first layer will be the air layer.

The particles 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 corresponds to the bottoms of the particles. The in-plane distribution of particles is linked with the way the particles interfere with each other, which 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.

3.2.2 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`.

3.2.3 Programs

The examples presented in the next paragraphs are written in Python. For tutorials about this programming language, the users are referred to [**?**].

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 at the end of this section (Listing **??**).

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

```
1 import sys, os, numpy
2
3 from libBornAgainCore import *
```

First step: Defining materials

```
4 def RunSimulation():
5     #  defining materials
6     mAmbience = MaterialManager.getHomogeneousMaterial("Air",
          0.0, 0.0)
```

```
7     mSubstrate = MaterialManager.getHomogeneousMaterial("
          Substrate",
8     6e-6, 2e-8)
9     mParticle = MaterialManager.getHomogeneousMaterial("Particle"
          , 6e-4,
10  2e-8 )
```

Line 4 marks the beginning of the function to define and run the simulation.
Lines 6, 8 and 10 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 $1-6\times10^{-6}+i2\times10^{-8}$, and the material of particles, whose refractive index is n= $1-6\times10^{-4}+i2\times10^{-8}$.

Second step: Defining the particles

```
11    # collection of particles
12    cylinder_ff = FormFactorCylinder(5*nanometer, 5*nanometer)
13    cylinder = Particle(mParticle, cylinder_ff)
14    prism_ff = FormFactorPrism3(5*nanometer, 5*nanometer)
15    prism = Particle(mParticle, 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 12). Then we associate this shape with the constituting material as in line 13.

The same procedure has been applied for the prism in lines 14 and 15 respectively.

Third step: Characterizing the layers and assembling the sample

Particle decoration

```
16    particle_decoration = ParticleDecoration()
17    particle_decoration.addParticle(cylinder, 0.0, 0.5)
18    particle_decoration.addParticle(prism, 0.0, 0.5)
19    interference = InterferenceFunctionNone()
20    particle_decoration.addInterferenceFunction(interference)
```

The object which holds the information about the positions and densities of particles in our sample is called `ParticleDecoration` (line 16). We use the associated function `addParticle` for each particle shape (lines 17, 18). Its general syntax is

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

where `<particle_name>` is the name used to define the particles (lines 13 and 15), `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 particles in a layer are given in relative coordinates. For the top layer, the bottom 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 19 and 20 specify that there is **no coherent interference** between the waves scattered by these particles. The intensity is calculated by the incoherent sum of the scattered waves: $\langle |F_n|^2 \rangle$, where F_n is the form factor associated with the particle of type n. 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 17 and 18.

Multilayer

```
21    # air layer with particles and substrate form multi layer
22    air_layer = Layer(mAmbience)
23    air_layer.setDecoration(particle_decoration)
24    substrate_layer = Layer(mSubstrate, 0)
25    multi_layer = MultiLayer()
26    multi_layer.addLayer(air_layer)
27    multi_layer.addLayer(substrate_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 22, we use the previously defined `mAmbience` (="air" material) (line 6). The command written in line 23 shows that this layer is decorated by adding the particles using the function `particle_decoration` defined in lines 16-20. The substrate layer only contains the substrate material (line 24).

There are different possible syntaxes to define a layer. As shown in lines 22 and 24, 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 25): we start with the air layer decorated with the particles (line 26), which is the layer at the top and end with the bottom layer, which is the substrate (line 27).

Fourth step: Characterizing the input beam and output detector and running the simulation

```
28      # run simulation
29      simulation = Simulation()
30      simulation.setDetectorParameters(100,-1.0*degree, 1.0*degree,
31                                       100, 0.0*degree, 2.0*degree, True
                                            )
32      simulation.setBeamParameters(1.0*angstrom, 0.2*degree, 0.0*
            degree)
33      simulation.setSample(multi_layer)
34      simulation.runSimulation()
```

The first stage is to define the `Simulation()` object (line 29). Then we define the detector (line 31) and beam parameters (line 32), which are associated with the sample previously defined (line 33). Finally we run the simulation (line 34). 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: Note that, except for `isgisaxs_style`, there are no default values implemented for the parameters of the beam and detector.

Line 34 shows the command to run the simulation using the previously defined setup.

Fifth step: Saving the data

```
35     # retrieving intensity data
36     return GetOutputData(simulation)
```

In line 36 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. Some options are provided by `BornAgain`. For example, figure 3.2 shows the two-dimensional contourplot of the intensity as a function of α_f and ϕ_f.

3.4 Example 2: working with sample parameters.

This section gives additional details about the manipulation of sample parameters at run time, that is after the sample has already been constructed. For single simulation this is normally not necessary, however it might be useful during interactive work when user tries to find optimal sample parameters by running a series of consequent simulations. Similarly, this task arises when the theoretical model, presented by the sample and the simulation descriptions, is used for the fitting of real data. In this case fitting kernel has to be informed about existing sample parameters and has to have a mechanism for changing values of these parameters to find they optimal values.

In `BornAgain` this is done using so called sample parameter pool mechanism and we will briefly explain it using example from the previous Section 3.3.

Inside `BornAgain` sample is described by a hierarchical tree of objects. For the multilayer created in previous section this tree can be graphically represented as shown in Fig. 3.3. Similar tree can be printed in 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`. Children objects by turn might also be composed into tree-like structure. For example, `Layer #0` contains `ParticleDecoration` object which holds

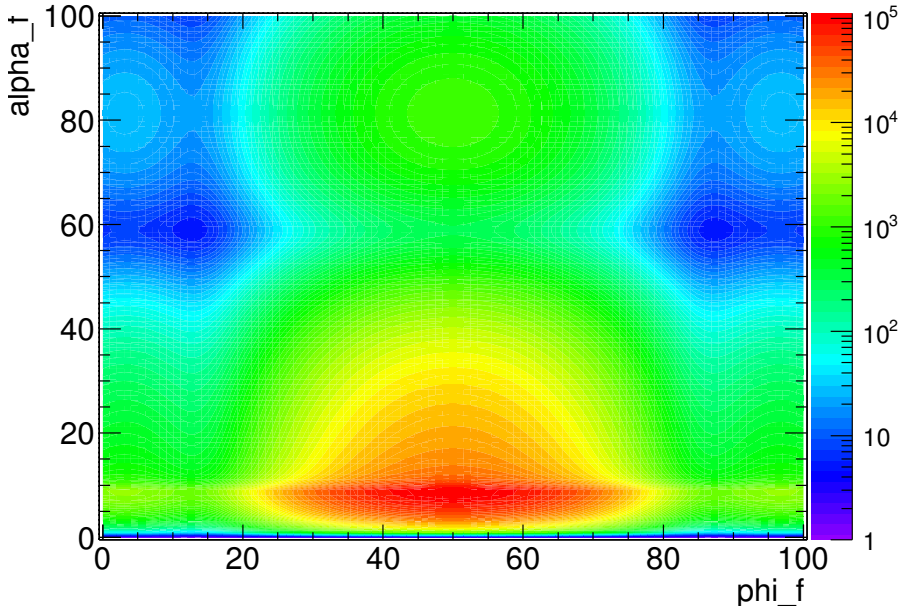

Figure 3.2: Figure of example 1: Simulated grazing-incidence small-angle X-ray scattering from a mixture of cylindrical and prismatic nanoparticles without any interference, deposited on top of a substrate. The input beam is characterized by a wavelength λ of 1 Å and incident angles $\alpha_i = 0.2°$, $\phi_i = 0°$. 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.

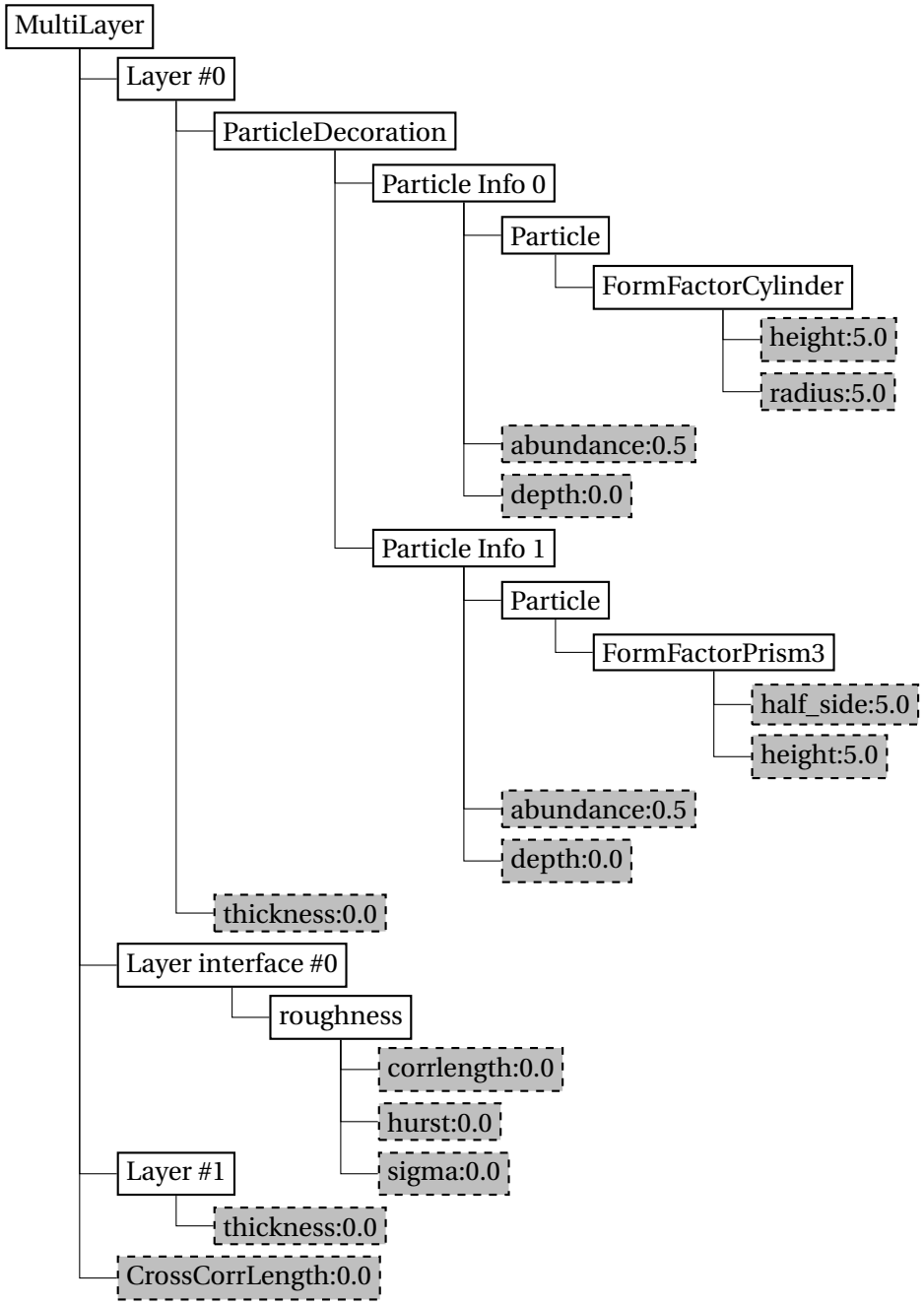

Figure 3.3: Tree representation of the sample structure.

information related to the two types of particles populating the layer. All numerical values which have been used during sample construction (thickness of layers, size of particles, roughness parameters) are the 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 from the names of corresponding nodes of the tree and can be accessed/changed during run time. For example, the `height` of the cylinders populating first layer can be changed from current 5 nm to 1 nm by running the command

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

The user can get names and values of all registered sample's parameters 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
'/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 group of parameters. In example below first command will change the height of the cylinders in the same way, as in previous example, while 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 to 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 following chapter.

The chapter starts from the short introduction to the basic concept of data fitting in Section 4.1. If user is familiar with it, he is welcome to proceed to the Section 4.2 containing details of the implementation in `BornAgain` . Advanced fitting techniques, including fine tuning of minimization algorithms, simultaneous fit of different data sets, parameters correlation, are covered in Section 4.4. Section 4.3 contains Python fitting example with detailed explanations of every fitting step. Section 4.5 contains a few practical advises which might help the user to get right answers from `BornAgain` fitting.

4.1 Gentle introduction to the data fitting.

The aim of this section is to briefly introduce the basic concept of data fitting, its key terminology and difficulties which might arise in scattering data fit. Users wanting to find out more about minimization (also called maximization or optimization methods depending on the formulations and objectives) or looking for more rigorous discussion than provided in this manual are referred to [**?**, **?**]

4.1.1 Toy scattering experiment.

Fig. 4.1,left shows scattering intensity map in arbitrary units as a function of (x,y) of the detector "measured" in toy scattering experiment.

Scattering picture presented reminds some of GISAS patterns, nevertheless it is generated using simple function

$$I(x,y) = G(0.1,\ 0.01) + \frac{sin(x)}{x} \cdot \frac{sin(y)}{y}$$

Here $G(0.1, 0.01)$ is a random variable distributed according to the Gaussian distribution

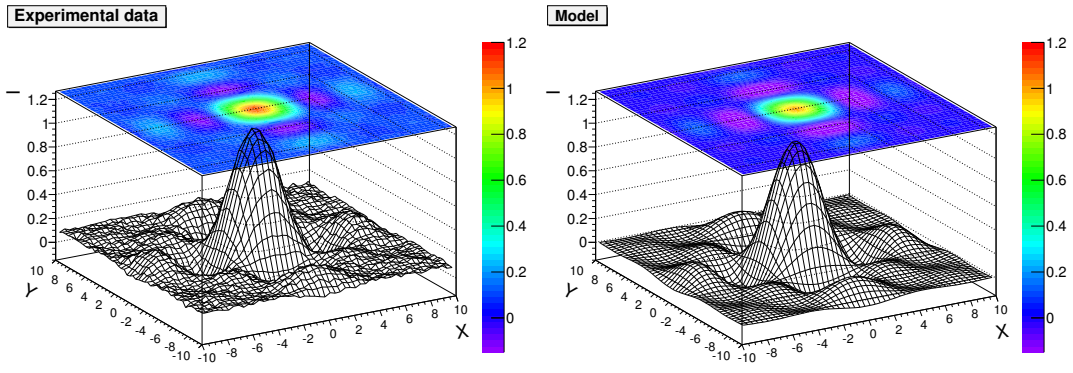

Figure 4.1: Intensity as a function of (x,y) detector coordinates obtained from toy experiment (left) and from the toy simulation (right).

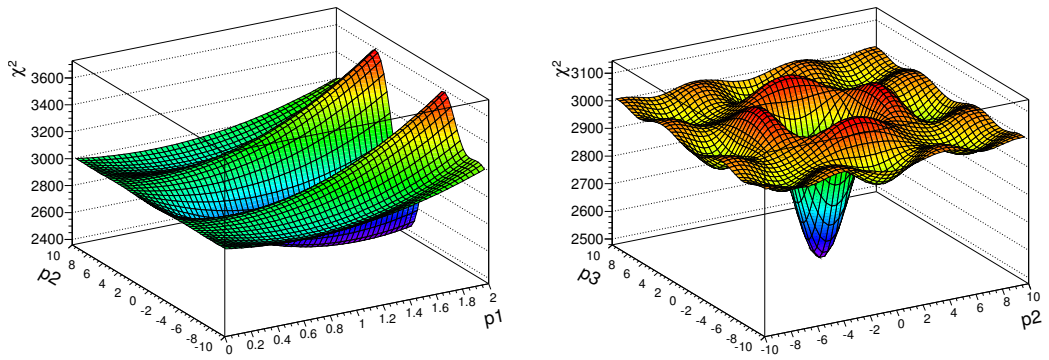

Figure 4.2: χ^2 value calculated between experimental and simulated data as a function of p_1, p_2 parameters (left) or p_2, p_3 parameters (right) used in the model.

with mean 0.1 and $\sigma = 0.01$. Constant 0.1 symbolize our experimental background and constant 0.01 is referred to the detector noise. The rest of the formula represents our signal.

Lets define our model, namely specific mathematical function, to which we will fit our toy experimental data. By making an educated guess we assume that scattering intensity observed in the experiment should be described with the help of $sinc$ functions as follows

$$f(x,y;p) = p_0 + p_1 \cdot sinc(x-p_2) \cdot sinc(y-p_3) \tag{4.1}$$

The model has four parameters: p_0 describing background, p_1 describing signal strength and p_2, p_3 responsible for the peak position. Fig. 4.1,right shows the intensity as a function (x,y) calculated according our model using fixed parameter set $p_0 = 0, p_1 = 1, p_2 = 0, p_3 = 0$.

Two distributions look pretty much the same, however to find exact values of parameters which describe experimental data in the best way, one have to

- elaborate criteria for the difference between an actual data and its model
- employ minimization procedure which will minimize that difference

4.1.2 Objectives

The goal is to obtain the best fit of an observed distribution to a prediction by modifying a set of parameters from the prediction. This problem can be one or multi-dimensional and also linear or nonlinear. The quantity to minimize is often referred to as the *objective function*, whose expression depends on the particular method, like the maximum likelihood, the χ^2 minimization or the expected prediction error function.

χ^2 or least squares minimization

A dataset consist of n data pairs $(\mathbf{x_i}, a_i), i = 1, N$ where $\mathbf{x_i}$ is an independent variable and a_i is dependent variable, whose value is found in the measurement i. The number N denotes the total number of measurements.

In the case of intensity map measured in our toy experiment and presented in Fig 4.1, a variable a_i denotes measured intensity, a variable $\mathbf{x_i}$ is a vector and correspond to the (x_i, y_i) coordinates of pixels in our detector while number N corresponds to total number of detector pixels.

The model function has the form $f(\mathbf{x_i}, \mathbf{p})$ where adjustable parameters are held in the vector $\mathbf{p}$. The least squared method finds the optimum of model function which fit the data in the best way by searching for the minimum of the sum of squared residuals

$$\chi^2(\mathbf{p}) = \sum_{i=1}^{N} r_i^2$$

where residual is defined as the difference between measured value and the value predicted by the model.

$$r = a_i - f(\mathbf{x_i}, \mathbf{p})$$

In the case of normally distributed variables with the σ^2 variance the quantity to minimize becomes

$$\chi^2(\mathbf{p}) = \frac{1}{d}\sum_{i=1}^{N}\frac{(a_i - f(\mathbf{x_i}, \mathbf{p}))^2}{\sigma^2}$$

where $d = N - k$ is number of degree of freedom (k number of free fit parameters).

Maximum of likelihood

to be written

Minimization algorithms

There are a large number of minimization algorithms providing a solution to the problem of minimizing the objective function over the space of parameters of the function. The minimization starts from initial guess for the parameters provided by the user, and then evolves iteratively under control of minimization algorithm. The procedural modifications on the parameters, the objective function, as well as convergence criterion depend on the method implemented. Details of particular implementation is beyond the scope of this manual and interested reader is encouraged to look at outside resources.

Local minima trap

Finding the global minimum of objective function is a general problem in optimization that is unlikely to have an exact and tractable solution. The problem can be illustrated using our toy experiment.

The theoretical model given by the formula 4.1 is defined in (x, y) space and additionally depends on parameter vector $\mathbf{p}$. The χ^2 objective function is obtained by the calculation of sum of squared residuals between measured (Fig. 4.1, left) and predicted (Fig. 4.1, right) values over x, y space. It is defined in parameter space $\mathbf{p}$ which have 4 dimensions.

Fig. 4.2 (left) shows χ^2 as a function of p_1, p_2 parameters while parameters p_0, p_3 remain fixed. Fig. 4.2 (right) shows χ^2 as a function of p_2, p_3 parameters while parameters p_0, p_1 remain fixed.

One can see that given objective function have a strongly pronounced global minimum, the goal of our search, supplemented by a number of local minima. The later will provide a hostile environment for the minimization algorithm, causing poor or slow convergence to single global minimum.

4.1.3 Terminology.

Reference data
Normally just experimental data or might be also simulated data spoiled with the noise for purpose of testing of minimization algorithms.

Objective function

Subject of minimization procedure.

Minimization
Finding a best available values (i.e. local minimum) of some objective function.

Number of degrees of freedom
Number of data points - number of parameters in the fit.

Minimizer
An algorithm which minimize objective function.

4.2 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 using χ^2 or maximum likelihood methods. The features include

- Variety of multidimensional minimization algorithms and strategies.
- The choice over possible fitting parameters, they properties and correlations.
- Full control on χ^2 calculations, including application of different normalizations and assignment of different masks and weights to the different areas of reference data.
- The possibility to fit simultaneously an arbitrary number of data sets.

Fig. 4.3 shows general work flow of fitting procedure. Before running the fitting the

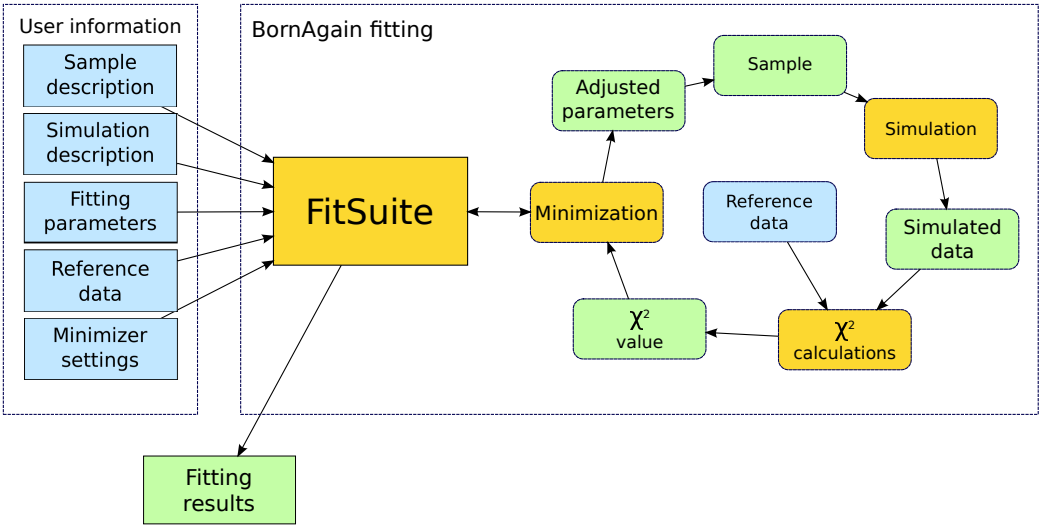

Figure 4.3: Fitting work flow.

user is required to prepare a number of data and to configure fitting kernel of `BornAgain` . Necessary stages consist of

- Preparing sample and simulation description (multilayer, beam, detector parameters).
- Choice of fitting parameters.
- Loading of reference data.
- Defining minimization settings.

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

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

Iteration process is going on without user intervention under the control of currently selected minimization algorithm. 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 to the user application fitting results can be retrieved. That consist of the best χ^2 value found, corresponding optimal sample parameters and 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 fitting procedure.

4.2.1 Preparing sample and 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.

4.2.2 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 parameter pool mechanism. That mechanism is explained in details in Section 3.4 and it is recommended to read it before proceeding further.

User specifies selected sample parameters as a fit parameter using `FitSuite` and its `addFitParameter` method

```
fit_suite = FitSuite()
fit_suite.addFitParameter(<name>, <value>, <AttLimits>)
```

Here the `<name>` correspond to the name of the parameter in the sample's parameter pool. By using wildcard's in the parameter name the group of sample parameters, corresponding to the given pattern, can be associated with single fitting parameter and fitted simultaneously to get common optimal value.

The second parameter <value> correspond to the initial value of fitting parameter while the third one `<AttLimits>` corresponds to the boundaries imposed on the range of variations of that 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.

4.2.3 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 is 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 ASCII file containing axes binning description and the intensity data itself.

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

To associate the simulation with the reference data the method `addSimulationAndRealData` has to be used as shown

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

here <simulation> correspond to the BornAgain simulation object with sample, beam and detector fully defined, <reference> correspond to the experimental data object obtained from ASCII file and <chi2_module> is an optional parameter for advanced control of $\chi 2$ calculations.

There is a possibility to call given method more than once to submit more than one pair of <simulation>, <reference> to the fitting procedure and so to provide simultaneous fit of some combined data set.

By using the third <chi2_module> parameter different normalization and weights can be applied to let the user fully control the way $\chi 2$ is calculated. This feature will be explained in Section 4.4.

4.2.4 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 setting 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 algorithms can also be obtained using MinimizerFactory.printCatalogue() command.

There are several options common for every minimization algorithms, which can be changed before minimization starts. They are handled by MinimizerOptions class:

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

In given code snippet a number of “maximum function calls”, namely a number of times the minimizer is allowed to call the simulation, is limited to the 10. The minimizer will take that number into consideration and will try to limit number of iterations by that value.

There is also a number of expert level options common for all minimizers as well as a number of possibilities to tune individual minimization algorithms. They will be explained in Section 4.4.

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

```
fit_suite.printResults()
```

Section 4.3 gives more details about access to fitting results.

4.3 Basic Python fitting example.

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

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

The example uses same sample geometry as in Section 3.3. It represents cylindrical and prismatic particles in equal proportion, in an air layer, deposited on a substrate layer, with no interference between the particles. We consider following parameters to be unkown

- the radius of cylinders
- the height of cylinders
- 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, fitting procedure is able to restore correct value of 5 nm for all parameters.

Importing Python libraries

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

We start from importing two `BornAgain` libraries required to create 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, 0)
29     multi_layer = MultiLayer()
30     multi_layer.addLayer(air_layer)
31     multi_layer.addLayer(substrate_layer)
32     return multi_layer
```

Function starting at the line 5 creates 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
```

Function starting at the line 35 creates simulation object with beam and detector parameters defined.

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 sample and simulation description and assign the sample to the simulation. Our reference data are contained in the file `'refdata_fitcylinderprisms.txt'`. In our case this reference had been generated by adding noise on the scattered intensity from a numerical sample with a fixed length of 5 nm of the four fitting parameters (*i.e.* the dimensions of the cylinders and prisms). Line 53 creates real data object by loading ASCII data from the 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 set up `FitSuite` to print on the screen the information about fit progress every 10^{th} iteration.

```
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 syntax of `addFitParameter` is

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

where `<name>` is the name of sample pool parameters (see Section 3.4) selected as a fitting parameter. Then we input its initial value and the iteration step used in the minimization process. Finally `<limits>` specify the boundaries of the parameter's value. Here 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 the boundaries are imposed only on the lower one of 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 to fit results.

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

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

4.4 Advanced fitting.

4.4.1 Affecting $\chi 2$ calculations.

4.4.2 Simultaneous fit of several data sets.

4.4.3 Using fitting strategies.

4.4.4 Masking the real data.

4.4.5 Tuning fitting algorithms.

4.4.6 Fitting with correlated sample parameters.

4.5 How to get right answer from fitting.

As it was already mentioned in Section 4.1, one of the main difficulties in fitting the data with the model is the presence of multiple local minima in the objective function. The extended list of problems causing fit to failure includes

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

Given list, of course, is unrelated only to `BornAgain` fitting. It remains the same while fitting the data with any fitting program and any kind of theoretical model. To address all these difficulties some amount of manual tuning might be necessary. Below we give some recommendations which might help the user to achieve reliable fit results.

General recommendation

- initially choose small number of free fitting parameters
- eliminate redundand parameters
- provide a good initial guess for fit parameters

- start from default minimizer settings and turn to the fine tuning after some experience has been acquired.
- repeat fit using different starting values for parameters or their limits
- repeat fit fixing and releasing different groups of parameters
- use `Minuit2` minimizer with `Migrad` algorithm (default) to get most reliable parameter error estimation
- try `GSLMultiFit` minimizer or `Minuit2` minimizer with `Fumili` algorithm to get fewer iterations

  to be continued...

Minimizer name	**Algorithm**	**Description**
`Minuit2` [**?**]	`Migrad`	According to [**?**] 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` [**?**]	`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` [**?**]		Levenberg-Marquardt Algorithm
`GSLSimAn` [**?**]		Simulated Annealing Algorithm

Table 4.1: List of minimizers implemented in `BornAgain`.

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 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 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 from a few external and well established open-source libraries: boost, GNU scientific library, Eigen and Fast Fourier Transformation libraries. They are required to be present on the system to run BornAgain on Unix Platform. In the case of Windows Platform they will be added to the system automatically during BornAgain installation. Other libraries shown on the plot (ROOT, matplotlib) are optional.

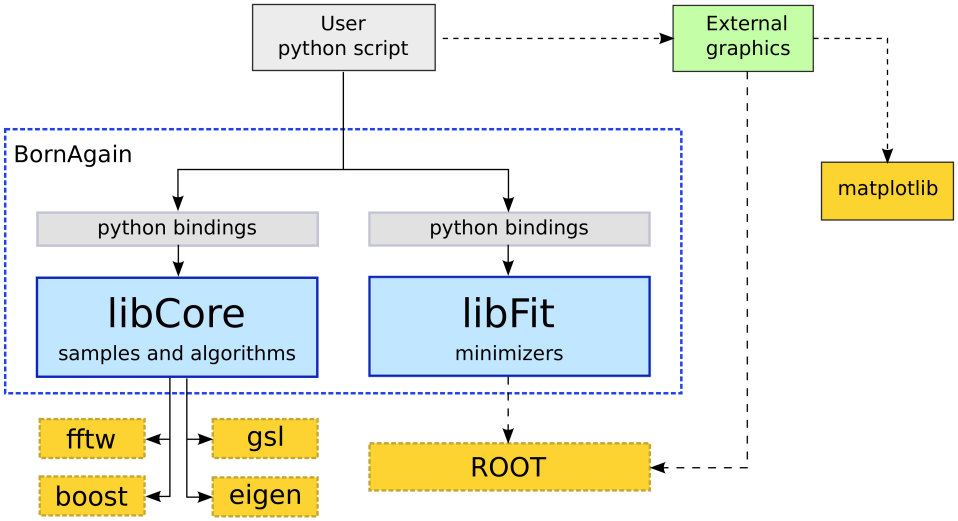

Figure 5.1: Structure of `BornAgain` libraries.

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 of 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.

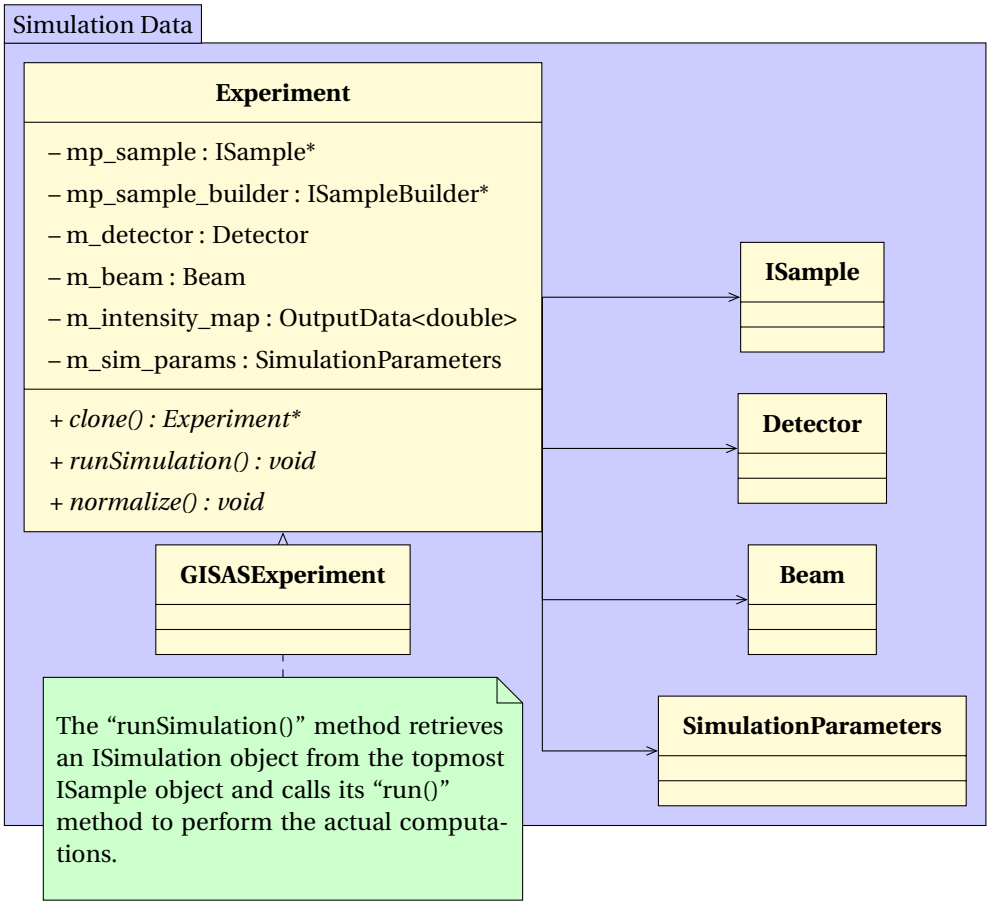

Figure 5.2: The Experiment class as a container for sample, beam, detector and simulation parameters.

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.

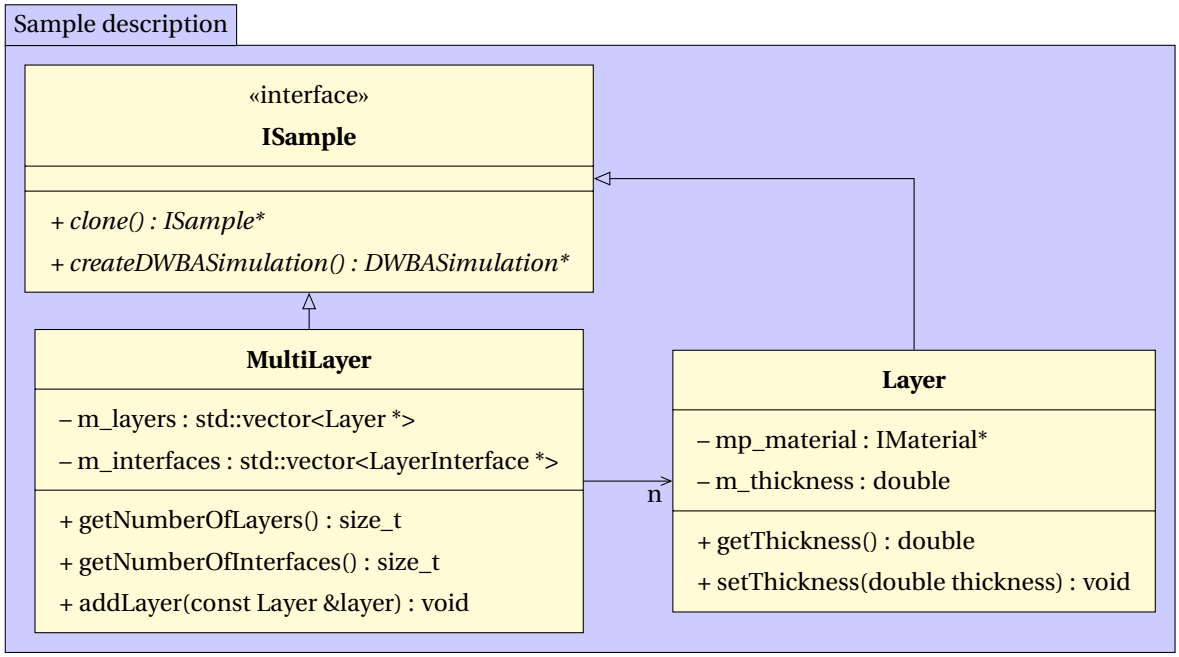

Figure 5.3: The ISample interface

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.

Script can be fount 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
           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 = GetOutputData(simulation) + 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.

Script can be fount 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
           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, 0)
29     multi_layer = MultiLayer()
30     multi_layer.addLayer(air_layer)
31     multi_layer.addLayer(substrate_layer)
32     return multi_layer
33
34
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
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()
```