BornAgain

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

User Guide
version 0.1.1
September 25, 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

Contents

Introduction **3**

1 Quick start **5**
1.1 Quick start on Unix Platforms 5
1.2 Quick start on Windows Platforms 6
1.3 Getting help 6

2 Installation **7**
2.1 Building and installing on Unix Platforms. 7
2.1.1 Third-party software. 8
2.1.2 Getting source code 9
2.1.3 Building and installing the code 10
2.1.4 Running first simulation 11
2.2 Installing on Windows Platforms. 11

3 Software architecture **12**
3.1 Data classes for simulations and fits 13
3.1.1 The Experiment object 13
3.1.2 The ISample class hiërarchy 14

4 Simulation **16**
4.1 General methodology 16
4.2 Conventions 16
4.2.1 Geometry of the sample 16
4.2.2 Units 17
4.2.3 Programs 17
4.3 Example 1: Two types of islands on top of substrate. No interference function 17

5 Fitting **25**
5.1 Short description of fitting theory 25
5.1.1 Objectives 25
5.1.2 How good is the fitting result? 26
5.1.3 Main features of the minimization algorithm 27
5.1.4 Terminology 28

5.2 Implementation in `BornAgain` 28
5.2.1 General fitting procedure 28
5.2.2 Example in Python 32

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 [**?**]. As novel features, BornAgain supports an unrestricted number of layers and particles, diffuse reflection from rough layer interfaces, and particles with inner structures. Support for polarized neutrons and magnetic scattering is forthcoming. 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 the simulation on Unix and Windows platforms in Section 1. Detailed description of installation procedure is given in Section 2. Section 3 provides brief overview of software architecture, while general methodology of simulation with BornAgain and detailed usage examples are given in Section 4. Fitting toolkit provided by the framework are

presented in Section 5.

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
Use git repository

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

or download `BornAgain` source tarball from Download section of `http://apps.jcns.fz-juelich.de/BornAgain`

Step III: building the source

```
mkdir <build_dir>; cd <build_dir>;
cmake <source_dir> -DCMAKE_INSTALL_PREFIX=<install_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

- python-2.7, matplotlib, numpy

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

Step IV: running example
Run example by double-click on python script located in `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 installation of the framework or in running the simulation can use a web based issue tracking system at `http://apps.jcns.fz-juelich.de/redmine/projects/bornagain/issues` to provide bug report. The same system can be used for enhancement requests. The system is open for all users in read mode, while submitting of bug reports and feature requests are possible after 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 (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)
- 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>.tgz
```

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 <source_dir> -DCMAKE_INSTALL_PREFIX=<install_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
Current version of `BornAgain` requires `Python, numpy, matplotlib` to be installed on the system. We recommend usage of `PythonXY` at `https://code.google.com/p/pythonxy` which contains all three packages bundled together. User has to download and install given package before proceeding with `BornAgain` installation.

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

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

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

Chapter 3

Software architecture

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

- User imports BornAgain libraries into the python to extent it with BornAgain API and then prepare python script with sample description and simulation settings. User runs the simulation by executing script in python interpreter and then assess simulation results using the way he likes, e.g. python + numpy + matplotlib.
- User may construct standalone C++ application linked to BornAgain libraries.
- User interacts with the framework through graphical user drag-and-drop interface (forthcoming).

Object oriented approach in the simulation design allows to reach much higher level of flexibility in sample construction, decouple interdependence in internal calculations and so facilitate creation of new models that entails little or no modification to the existing code.

The general structure of BornAgain and the way user interacts with it are shown in Fig. 3.1. The framework kernel consists of two shared libraries, libCore and libFit. Thanks to the python interface they can be imported to the python as external modules. The library libCore contains a number of classes, grouped into several class categories necessary for the description of model and running the simulation. The library libFit contains a number of minimization engines and interfaces to them, to let user to fit real data with the model defined.

BornAgain depends from a few external 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. Other libraries shown on the plot (ROOT, matplotlib) are optional.

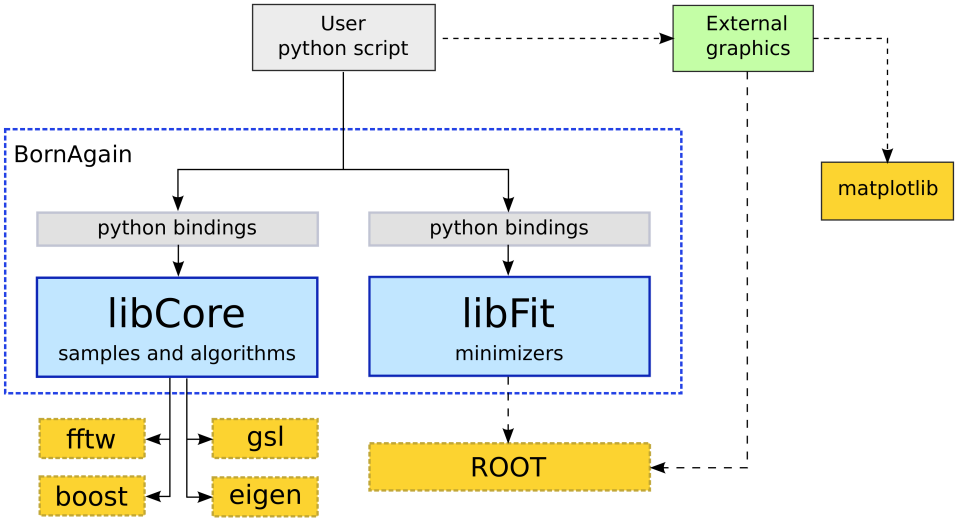

Figure 3.1: Structure of `BornAgain` libraries.

3.1 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?

3.1.1 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 3.2.

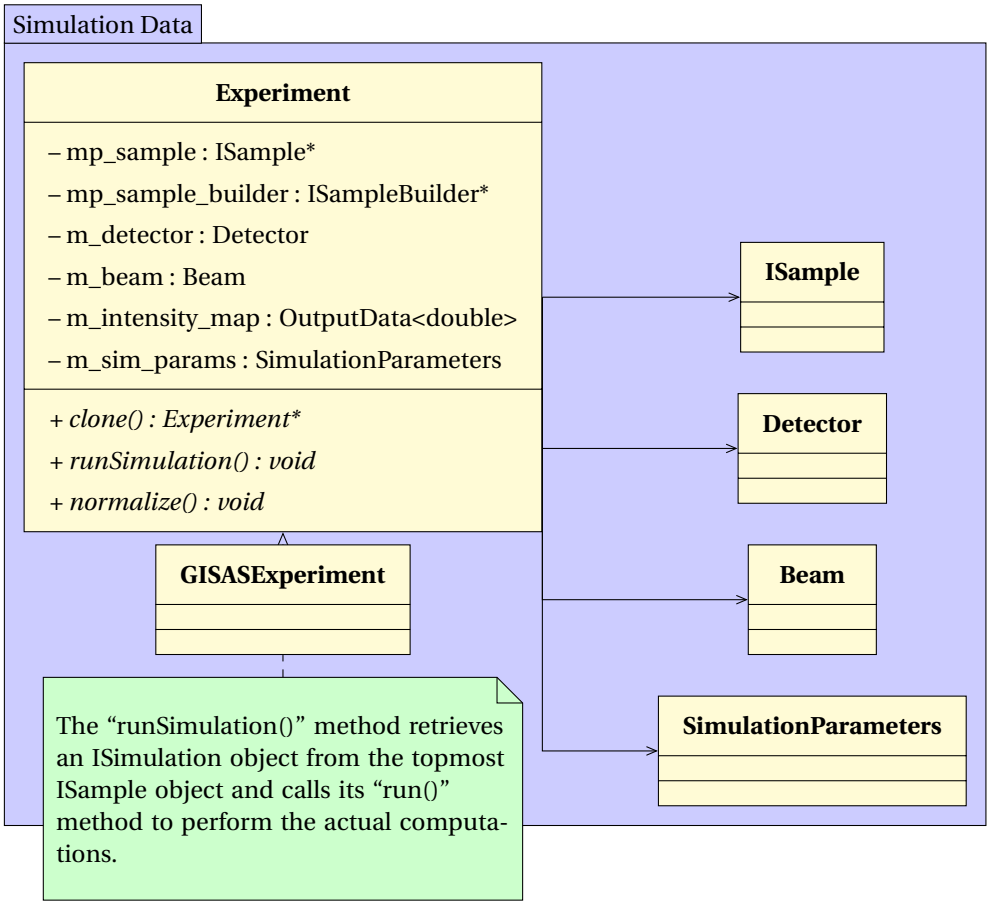

Figure 3.2: The Experiment class as a container for sample, beam, detector and simulation parameters.

3.1.2 The ISample class hiërarchy

Samples are described by a hiërarchy 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 in question. The coupling is however not very tight, 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 3.3.

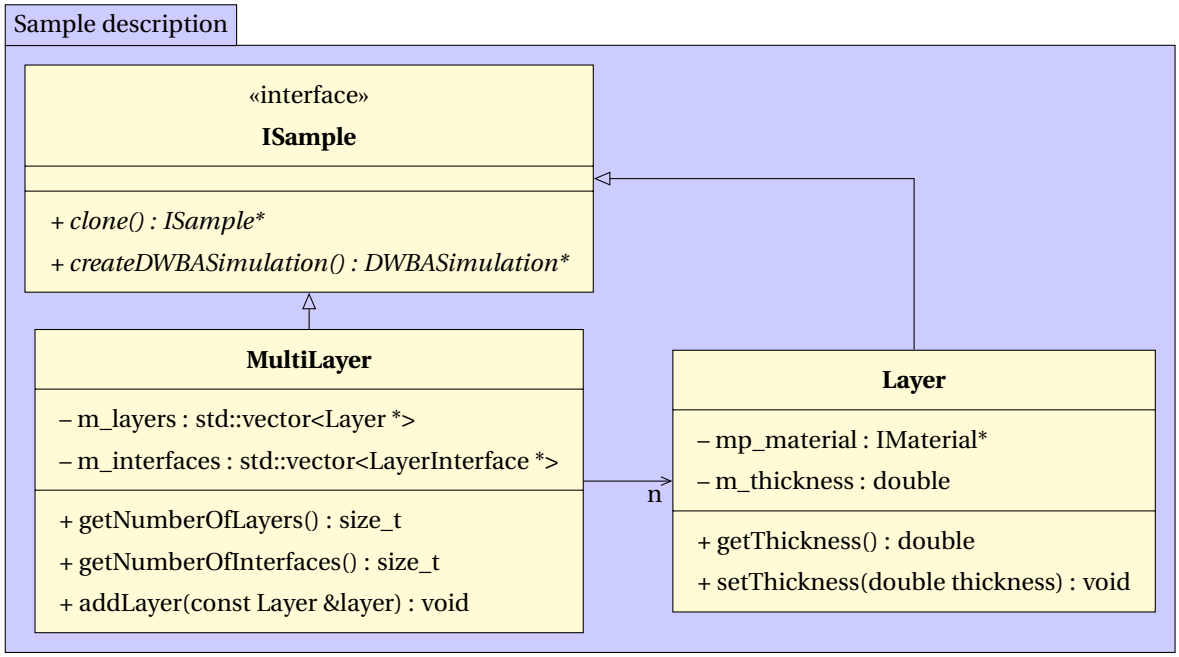

Figure 3.3: The ISample interface

Chapter 4

Simulation

4.1 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.

The sample is built from object oriented building blocks instead of loading data files.

4.2 Conventions

4.2.1 Geometry of the sample

The geometry used to describe the sample is shown in figure 4.1. 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 for the angles α_i and α_f is so that they are positive as shown in figure 4.1.

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.

4.2.2 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`.

4.2.3 Programs

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

4.3 Example 1: Two types of islands on top of substrate. No interference function

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

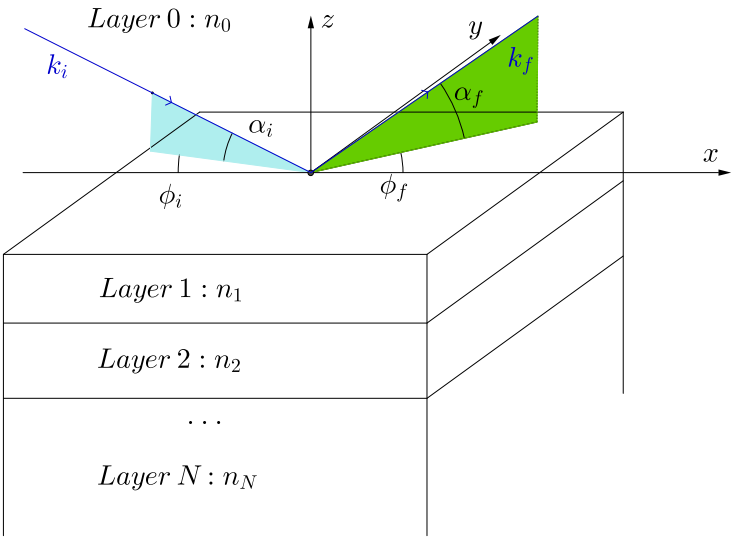

Figure 4.1: Representation of the scattering geometry. n_j is the refractive index of layer j and α_i and ϕ_i are the incident angle of the wave propagating. α_f is the exit angle with respect to the sample's surface and ϕ_f is the scattering angle with respect to the scattering plane.

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 4.1).

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 4.1 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 4.1.

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 4.2 shows the two-dimensional contourplot of the intensity as a function of α_f and ϕ_f.

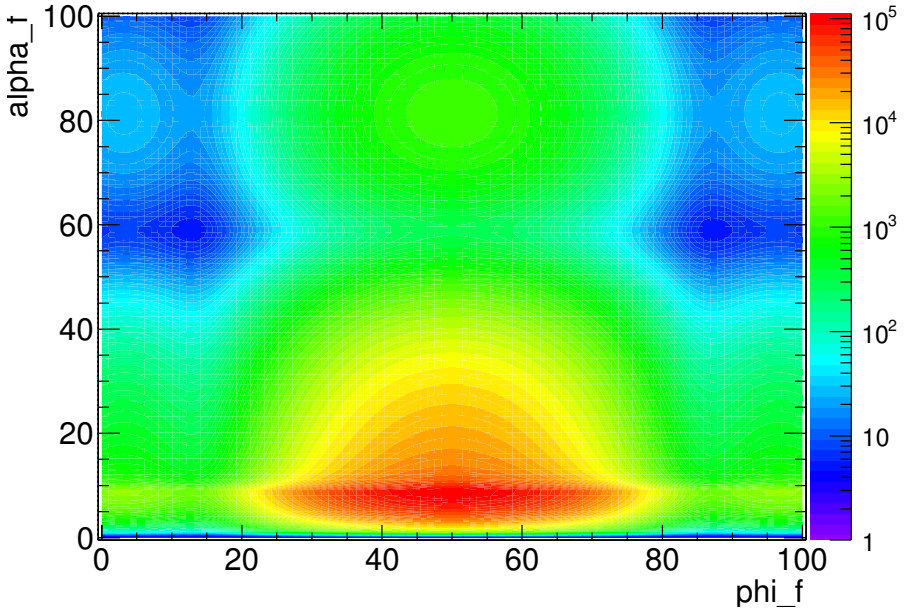

Figure 4.2: Figure of example 1: Simulated grazing-incidence small-angle X-ray scattering from a mixture of cylindrical and prismatic nanoparticles without any interference, deposited on top of a substrate. The input beam is characterized by a wavelength λ of 1 Å and incident angles $\alpha_i = 0.2°$, $\phi_i = 0°$. 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.

```
import sys, os, numpy

sys.path.append(os.path.abspath(os.path.join(os.path.split(
    __file__)[0],'..', '..', '..', 'lib')))

from libBornAgainCore import *

def RunSimulation():
    #  defining materials
    mAmbience = MaterialManager.getHomogeneousMaterial("Air",
        0.0, 0.0 )
    mSubstrate = MaterialManager.getHomogeneousMaterial("
        Substrate",
    6e-6, 2e-8)
    mParticle = MaterialManager.getHomogeneousMaterial("Particle"
        , 6e-4, 2e-8 )
```

```
    # collection of particles
    cylinder_ff = FormFactorCylinder(5*nanometer, 5*nanometer)
    cylinder = Particle(mParticle, cylinder_ff)
    prism_ff = FormFactorPrism3(5*nanometer, 5*nanometer)
    prism = Particle(mParticle, prism_ff)
    particle_decoration = ParticleDecoration()
    particle_decoration.addParticle(cylinder, 0.0, 0.5)
    particle_decoration.addParticle(prism, 0.0, 0.5)
    interference = InterferenceFunctionNone()
    particle_decoration.addInterferenceFunction(interference)
    # air layer with particles and substrate form multi layer
    air_layer = Layer(mAmbience)
    air_layer.setDecoration(particle_decoration)
    substrate_layer = Layer(mSubstrate, 0)
    multi_layer = MultiLayer()
    multi_layer.addLayer(air_layer)
    multi_layer.addLayer(substrate_layer)

    # build and run simulation
    simulation = Simulation()
    simulation.setDetectorParameters(100,-1.0*degree, 1.0*degree,
                                     100, 0.0*degree, 2.0*degree,
                                        True)
    simulation.setBeamParameters(1.0*angstrom, 0.2*degree, 0.0*
       degree)
    simulation.setSample(multi_layer)
    simulation.runSimulation()

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

Listing 4.1: Python script of example 1

Chapter 5

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.

5.1 Short description of fitting theory

The aim of this section is to briefly introduce the basic concept of minimization and its key terminology. Users wanting to find out more about minimization (also called maximization or optimization methods depending on the formulations and objectives) are referred to [**?**, **?**].

5.1.1 Objectives

We generally have 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. In many cases, a number of distinct functions may need to be minimized at once, for example, when samples are generated with respect to an independent variable (time, position,...). Minimization can be done according to different definitions of the norm. For example, using the Euclidean norm, this process is often called a least squares problem. Weights can also be added in order to emphasize important points and neglect uncritical ones.

Remark:

- The term "minimizing" means finding a **local** minimum of the objective function.
- The number of observations must greatly exceed the number of fitting parameters that are to be estimated.

We are now going to detail two methods by specifying the expression of the function to minimize: maximum of likelihood and χ^2 minimization.

Maximum of likelihood

This is a popular method for parameters' estimations because the maximum likelihood estimators are approximately unbiased and efficient for large data samples, under quite general conditions. We consider a random variable $\mathbf{x}$ (it could be a vector) distributed with a distribution function $f(\mathbf{x};\alpha)$. We assume $f(\mathbf{x};\alpha)$ to be known except for the parameter(s) α (which could be a vector as well). The expression of $f(\mathbf{x};\alpha)$ represents the hypothesized probability density function for the $\mathbf{x}$ variable. Then, by repeating the measurements N times, we sample $x_1,\ldots,x_N$ values. The method of maximum likelihood takes the estimators to be those values of α that maximize the likelihood function $\mathscr{L}$ as $\mathscr{L}(\alpha) = \prod_{i=1}^{N} f(x_i;\alpha)$. Since it is easier to deal with a sum, we usually minimize $-\ln(\mathscr{L})$.

χ^2 or least squares minimization

Given a set of observations $\{x_1,x_2,\ldots,x_n\}$, with expectation values $\{y_1(\alpha),\ldots,y_n(\alpha)\}$ and covariance matrix V (matrix element written $V_{i,j}$), then the set of parameter values $\hat{\alpha}$ which minimizes the quantity: $\chi^2 = \sum_{i,j}[x_i - y_j(\alpha)]V_{ij}[x_j - y_j(\alpha)]$ is called the Least Squares Estimate for α.

If the x_i are sampled from a normal distribution, then the least squares minimization is equivalent to the maximum likelihood method: the set of parameters α which maximize $\mathscr{L}$ is the same as those which minimize χ^2. In this case the expression of χ^2 becomes $\sum_{i=1}^{N}[x_i - y_i(\alpha)]^2/\sigma_i^2$, where σ_i^2 is the variance on $y_i(\alpha)$. Even if the observations are not normally distributed, the least squares minimization may be useful, in particular, if the distribution is approximately normal.

5.1.2 How good is the fitting result?

In general, a minimization process is intended to compare a reference (experimental observations) to some predictions (numerical models) dependent on a certain number of parameters. At the end of the minimization procedure, the user has to determine how close the estimated parameters are to the reference ones. The first step could be a visual check by plotting both of them. On the quantitative side, different quantities could be evaluated. The most common tests for goodness-of-fit are the χ^2 test, Kolmogorov test, Cramer-Smirnov-Von-Mises test, runs.

The reduced χ^2 is defined as the final sum of the squared residuals divided by the number of degrees of freedom (*number of datapoints - number of parameters in the fit*). The fit can be considered as good if the reduced $\chi^2 \sim 1$.

Remark:

- A bad fit does not necessarily produce large errors and having a "too good" goodness-of-fit usually means that something is not right: for example, overestimated errors or the assumption of independent data when they were in fact correlated.
- The χ^2 test does not check that the uncertainties are Gaussian or normally distributed; it assumes that they are Gaussian. If the observations are not normally distributed, the LSE may be useful, in particular, if the distribution is approximately normal.

5.1.3 Main features of the minimization algorithm

We start with some initial guesses for the parameters. We can then proceed in different ways in order to find the best estimates of a local minimum of our objective function. The procedural modifications on the parameters, the objective function, as well as the convergence criterion depend on the method implemented. For example, the minimization could stop if the modifications on the objective function or on the parameters between consecutive iterative steps are lower than a given minimization tolerance.
The minimization algorithms can be classified into different categories:

- *search method*: the solution is obtained by using only function evaluations at different points by modifying, at each iteration, the interval between which the minimum is searched for.
- *approximate method*: close to a minimum, the function to minimize is approximated to a polynomial. The degree of approximation might require the evaluation of gradients or Hessian matrices (matrix of second-order partial derivatives of a function)

Many refinements and particularities as well as other algorithms' classifications exist. For example the minimization can be performed by using sequential search directions that keep track of the previous steps.

In addition to the objective function, there might be some additional conditions imposed on the parameters, for example, boundary conditions imposed on some variables or some extra relations between others. In this case, the problem is said to be a *constrained minimization problem*. Constraints make the process more technically challenging than in unconstrained situations.

5.1.4 Terminology

- number of degrees of freedom = number of data points - number of fitting parameters.
- The Hessian matrix or Hessian is a square matrix of second-order partial derivatives of a function. It describes the local curvature of a function of many variables.

5.2 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. These features include different multidimensional minimization algorithms and strategies from Root (Minuit2 and GSL libraries) and the choice over possible fitting parameters. The related codes are contained in the folder "Fit" (a detailed description is given in . . .).

5.2.1 General fitting procedure

The general fitting procedure can be split into different steps:

1. Creation of the sample: multilayered sample, beam, detector,
2. Choice of the parameters to fit,
3. Loading reference data,
4. Fit:
    - linking the reference and the numerical data,
    - choice of a minimizing algorithm (method, weights, strategy),
    - running the minimization,
    - checking the results.

The class FitSuite contains the main functionalities to be used for the fit. The following parts of this paragraph will detail each of the main stages before applying them to an example (see paragraph 5.2.2).

Building the sample

This step is similar for any simulation using BornAgain. 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.

Loading reference data

These are the data to which the fitting model will be compared to. They usually refer to experimental data. We assume that it is a two-dimensional intensity matrix as function of the output scattering angles α_f and ϕ_f (see Fig. 4.1). The user is required to provide reduced and **normalized** data.

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. These selected parameters represent the variables on which the minimizer will operate.

In `BornAgain`, the parameters used for the fit are specified using the function `addFitParameter` with the following list of variables: `(<name>, <value>, <step>, <AttLimits>, <error>)` where `<value>`, `<step>` and `<error>` are double values corresponding to the initial value of the parameter, the iteration step (optional parameter equal to 0.01 by default) and the error respectively. By default the input value of `<error>` is 0. `<AttLimits>` corresponds to the boundaries imposed on the range of variations of the fitting parameter's value. It can be

- `fixed()`,
- `lowerLimited(<min_value>)`,
- `limited(<min_value>, <max_value>)`.

where `<min_value>` and `<max_value>` are double values corresponding to the lower and higher boundary respectively. The unit of `<AttLimits>` is identical to the one used to characterize the parameter's `<value>`.

`<name>` is the reference to the parameter as it had been registered using `RegisterParameter`.
For example, to add the beam intensity to the list, `<name>` would be `"*Beam/intensity"`.
In the case of the cylindrical particles's height, it would become `"*FormFactorCylinder/height"`.

If the sample contains different types of particles, the heights of different particles can be associated to two different fitting parameters and minimized separately.

Hints:

- initially choose a small number of fitting parameters.
- provide a "good" initial guess to save time and reduce the risk of failure to find the minimum looked for.

Associating reference and numerical data

The minimization procedure deals with a pair of experimental data (the reference) and numerical data associated with function `addSimulationAndRealData`. This provides the function to minimize using the following syntax:

```
addSimulationAndRealData(<simulation>, <reference>, <chi2_module
    >)
```

where `<chi2_module>`, linked to the evaluation of χ^2 is optional. Its default implementation is $\chi^2 = (\text{simulation} - \text{reference})^2/\max(\text{reference}, 1)$. Other evaluations are possible using function
`setChiSquaredFunction` with the following parameter:

- `SquaredFunctionWithSystematicError(`ϵ`)` uses

$$\chi^2 = \frac{(\text{sim} - \text{reference})^2}{\max(|\text{reference}| + \epsilon^2 \text{reference}^2, 1)},$$

  where ϵ gives the ratio of systematic errors and is equal to 0.08 by default,

- `SquaredFunctionWithGaussianError(`σ`)` uses

$$\chi^2 = \frac{(\text{simulation} - \text{reference})^2}{\sigma^2},$$

  where σ is reference standard error and it is equal to 0.01 by default.

By default, all datapoints have the same weight of 1.

The users can therefore run a series of fits by changing this particular association between a numerical model and some experimental observations. For example, it is possible to generate a batch of different numerical samples by playing with the number of layers or the shapes of particles in order to obtain the best fit with the experimental data. It is possible to crop and select a single specific area in the two-dimensional space but not several isolated parts like around, for example, intensity maxima.

Choice of fitting method

Different minimizers from `Root` library can be used in `BornAgain`. They are listed in Table 5.1. Users can also add their own by implementing the appropriate definition in the `Catalogue` contained in program `MinimizerFactory`. `Minuit` user's manual describes which minimizer to use in order to best fit your data [**?**].

The list of minimizers implemented in `BornAgain` can be printed out using the command `print MinimizerFactory.print_catalogue()` for example in `Python`.

A particular algorithm is selected using function `setMinimizer`, whose syntax is the following:

```
setMinimizer(MinimizerFactory.createMinimizer("<Minimizer
name>","<optional algorithm>") )
```

where `<Minimizer name>` and `<optional algorithm>` can be chosen from the first and second column of Table 5.1 respectively. For example the users could select (‘‘`Minuit2`’’,’’`Migrad`’’) or (‘‘`GSLMultiFit`’’,`""`).

Some of these algorithms require the estimation of a gradient function associated with the function to minimize. `BornAgain` **implements it automatically if required**.

Remark: There is no default minimizer implemented in `BornAgain`.

Four strategies have been implemented in `BornAgain` and can be added using the function `addFitStrategy`:

- `FitSuiteStrategyDefault` is the default fit strategy. It just lets `FitSuite` run its minimization round,
- `FitSuiteStrategyAdjustData` adjusts the data before running the minimization round,
- `FitSuiteStrategyAdjustParameters` fixes fit parameters and then calls minimizer,
- `FitSuiteStrategyBootstrap` helps the minimizer get out of local minima by perturbing real data.

These strategies act on the parameters or the data and they are therefore different from those implemented in `Minuit2` [**?**], which are linked with how the minimizer runs. These `Minuit` strategies can be equal 0, 1 or 2 (default value =1). The smaller values are associated with fewer functions calls. On the contrary the higher values are more precise. In `BornAgain` we used the default value of 1.

Outputs

The minimization stops

- when the maximum number of function calls has been exceeded,
- 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
- if there had been a problem with the calculation of the covariance matrix.

The output of the minimization can be saved in a file or printed on the screen using the function `printResults()`. During the fitting process, intermediate results are accessible with function `initPrint(<print_every_nth>)`, where `<print_every_nth>` is the of the number of minimization iterations between outputs.

Special attention must be payed to the interpretation of the errors given by Minuit2 (normalization, reliability of the estimates determined by the minimizer, statistical interpretations) [**?**, **?**]. According to Minuit's documentation, "the best way to be absolutely sure of the errors, is to use "independent" calculations and compare them".

5.2.2 Example in Python

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. In addition, the complete listing of the script is given at the end (see Listing 5.1).

This example uses a simple sample geometry: cylindrical and prismatic particles in equal proportion, in an air layer, deposited on a substrate layer, with no interference between the particles.
We consider four fitting parameters: the radius and height of cylinders and the side length and 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 nanometers for the height of both particle shapes as well as for the radius of the cylinders and the half side length of the prisms' triangular basis.
Then we run our minimization consequently using the algorithm Migrad from Minuit2 as the minimization engine, 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.

Order of steps The stages concerned with the preparation of the fit (generation of the sample, characteristics of the input beam and output detector, loading of reference data) can be interchanged.

Importing Python libraries and defining parameters

```
1  import sys, os, numpy
2  import math
3
4  from libBornAgainCore import *
5  from libBornAgainFit import *
6
7  # values we want to find
8  cylinder_height = 5.0*nanometer
9  cylinder_radius = 5.0*nanometer
10 prism3_half_side = 5.0*nanometer
11 prism3_height = 5.0*nanometer
```

Apart from the standard Python libraries, we start by importing different libraries required in order to run the script. Lines 4 and 5 import two BornAgain libraries respectively linked with the generation of the sample and with the fitting. Then we specify the values

that our fitting parameters should be equal to at the end of the minimization (see lines 8-11).

Building the sample

```
12 # ----------------------------------
13 # create sample : cylinders and prisms in the air on substrate
       layer
14 # ----------------------------------
15 def buildSample():
16     # defining materials
17     mAmbience = MaterialManager.getHomogeneousMaterial("Air",
           0.0, 0.0 )
18     mSubstrate = MaterialManager.getHomogeneousMaterial("
           Substrate", 6e-6, 2e-8 )
19     mParticle = MaterialManager.getHomogeneousMaterial("Particle"
           , 6e-4, 2e-8 )
20     # collection of particles
21     cylinder_ff = FormFactorCylinder(cylinder_height,
           cylinder_radius)
22     cylinder = Particle(n_particle, cylinder_ff)
23     prism_ff = FormFactorPrism3(prism3_height,  prism3_half_side)

24     prism = Particle(n_particle, prism_ff)
25     particle_decoration = ParticleDecoration()
26     particle_decoration.addParticle(cylinder, 0.0, 0.5)
27     particle_decoration.addParticle(prism,0.0, 0.5)
28     interference = InterferenceFunctionNone()
29     particle_decoration.addInterferenceFunction(interference)
30     # air layer with particles and substrate form multi layer
31     air_layer = Layer(mAmbience)
32     air_layer.setDecoration(particle_decoration)
33     substrate_layer = Layer(mSubstrate, 0)
34     multi_layer = MultiLayer()
35     multi_layer.addLayer(air_layer)
36     multi_layer.addLayer(substrate_layer)
37     return multi_layer
38 # ----------------------------------
39 # create sample: input beam and detector - characteristics
40 # ----------------------------------
41 def createSimulation():
42     simulation = Simulation()
43     simulation.setDetectorParameters(100, 0.0*degree, 2.0*degree
           ,100 , 0.0*degree, 2.0*degree)
44     simulation.setBeamParameters(1.0*angstrom, 0.2*degree, 0.0*
           degree)
45     return simulation
```

The details about the generation of this multilayered sample and the characterization of the input beam and detector are given in Section 4.3. The only difference can be seen in lines 21, 23, where in this fitting example, we have to use names for the fitting parameters instead of numerical values.

Loading reference data

```
46 def GetRealData():
47     real_data = OutputDataIOFactory.getOutputData('
           Refdata_fitcylinderprisms.txt')
48     return real_data
```

Our reference data are contained in file `'Refdata_fitcylinderprisms.txt'`. They are expressed as a two-dimensional array of the output intensity as a function of α_f and ϕ_f (*i.e.* the two output scattering angles). 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).

Preparing the fitting pair

```
49 def run_fitting():
50     sample = buildSample()
51     simulation = createSimulation()
52     simulation.setSample(sample)
53     # get the reference data
54     real_data = GetRealData()
55     # run the simulation
56     simulation.runSimulation()
57     # linking reference and numerical (to be fitted) data
58     fitSuite = FitSuite()
59     fitSuite.addSimulationAndRealData(simulation, real_data)
```

Lines 50-56 generate the numerical model and load the reference data. Then with the `FitSuite` class, we associate this pair with `addSimulationAndRealData` (line 59).

Remark: `run_fitting()` function (line 49) is concerned with the complete fitting procedure: preparing the fitting pair but also the next states of choosing the fitting parameters, the minimizer, and running the fit. Therefore in Python, there must be an indentation for the script of the next stages. This point is made clearer at the end of this section where the full script is displayed.

Choice of fitting minimizer

```
60     fitSuite.setMinimizer( MinimizerFactory.createMinimizer("
           Minuit2","Migrad") )
```

Line 60 implements your choice of minimizer for the fit using the function `setMinimizer`. Several options are available in `BornAgain`; they are listed in Table 5.1

Choice of numerical parameters to be fitted

```
61    fitSuite.addFitParameter("*FormFactorCylinder/height", 4.*
          nanometer, 0.01*nanometer, AttLimits.lowerLimited(0.01) )
62    fitSuite.addFitParameter("*FormFactorCylinder/radius", 6.*
          nanometer, 0.01*nanometer, AttLimits.lowerLimited(0.01) )
63    fitSuite.addFitParameter("*FormFactorPrism3/height", 4.*
          nanometer, 0.01*nanometer, AttLimits.lowerLimited(0.01) )
64    fitSuite.addFitParameter("*FormFactorPrism3/half_side", 6.*
          nanometer, 0.01*nanometer, AttLimits.lowerLimited(0.01) )
```

Lines 61-64 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 the registered parameter 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.

⚠ Order of addition of fitting parameters The fitting parameters are stored in the order they are initialized. They can be accessed from the array `fitSuite.getFitParameters().getValues()` indexed from 0.

Running the fit

```
65    # run fit
66    fitSuite.runFit()
67    # print fit results
68    fitSuite.printResults()
```

Line 66 shows the command to start the minimization process. For this example we chose to display the final results only using the function `printResults()` (see line 68). But intermediate results are accessible as mentioned above with the command `printLine(<number of minimization iterations between prints>)`.

After running the fit, whose script is shown in Listing 5.1, the text given in 5.2 should be displayed on your screen (generated using `PrintResults`).

```
import sys, os, numpy
import math

sys.path.append(os.path.abspath(
                os.path.join(os.path.split(__file__)[0],
                '..', '..', '..', 'lib')))

from libBornAgainCore import *
from libBornAgainFit import *

# values we want to find
cylinder_height = 5.0*nanometer
cylinder_radius = 5.0*nanometer
prism3_half_side = 5.0*nanometer
prism3_height = 5.0*nanometer
# ----------------------------------
# create sample : cylinders and prisms in the air on substrate
    layer
# ----------------------------------
def buildSample():
    # defining materials
    mAmbience = MaterialManager.getHomogeneousMaterial("Air",
        0.0, 0.0 )
    mSubstrate = MaterialManager.getHomogeneousMaterial("
        Substrate",
    6e-6, 2e-8 )
    mParticle = MaterialManager.getHomogeneousMaterial("Particle"
        , 6e-4, 2e-8 )
    # collection of particles
    cylinder_ff = FormFactorCylinder(cylinder_height,
        cylinder_radius)
    cylinder = Particle(mParticle, cylinder_ff)
    prism_ff = FormFactorPrism3(prism3_height,  prism3_half_side)
    prism = Particle(mParticle, prism_ff)
    particle_decoration = ParticleDecoration()
    particle_decoration.addParticle(cylinder, 0.0, 0.5)
    particle_decoration.addParticle(prism,0.0, 0.5)
    interference = InterferenceFunctionNone()
    particle_decoration.addInterferenceFunction(interference)
    # air layer with particles and substrate form multi layer
    air_layer = Layer(mAmbience)
    air_layer.setDecoration(particle_decoration)
    substrate_layer = Layer(mSubstrate, 0)
    multi_layer = MultiLayer()
    multi_layer.addLayer(air_layer)
    multi_layer.addLayer(substrate_layer)
    return multi_layer
```

```
# ----------------------------------
# create sample : input beam and detector - characteristics
# ----------------------------------
def createSimulation():
    simulation = Simulation()
    simulation.setDetectorParameters(100, 0.0*degree, 2.0*degree
        ,100 , 0.0*degree, 2.0*degree)
    simulation.setBeamParameters(1.0*angstrom, 0.2*degree, 0.0*
        degree)
    return simulation
# ----------------------------------
# read "real" data from file
# ----------------------------------
def GetRealData():
    real_data = OutputDataIOFactory.getOutputData('
        Refdata_fitcylinderprisms.txt')
    return real_data
# ----------------------------------
# run fitting
# ----------------------------------
def run_fitting():
    sample = buildSample()
    simulation = createSimulation()
    simulation.setSample(sample)
    # get the real data, which is simply results of our
        simulation with default values
    real_data = GetRealData()
    # run the simulation
    simulation.runSimulation()
    # linking real and numerical (to be fitted) data
    fitSuite = FitSuite()
    fitSuite.addSimulationAndRealData(simulation, real_data)
    # setting fitting minimizer
    fitSuite.setMinimizer( MinimizerFactory.createMinimizer("
        Minuit2","Migrad") )
    # setting fitting parameters
    fitSuite.addFitParameter("*FormFactorCylinder/height", 4.*
        nanometer, 0.01*nanometer, AttLimits.lowerLimited(0.01) )
    fitSuite.addFitParameter("*FormFactorCylinder/radius", 6.*
        nanometer, 0.01*nanometer, AttLimits.lowerLimited(0.01) )
    fitSuite.addFitParameter("*FormFactorPrism3/height", 4.*
        nanometer, 0.01*nanometer, AttLimits.lowerLimited(0.01) )
    fitSuite.addFitParameter("*FormFactorPrism3/half_side", 6*
        nanometer, 0.01*nanometer, AttLimits.lowerLimited(0.01) )
    # run fit
    fitSuite.runFit()
    # print fit results
    fitSuite.printResults()
```

Listing 5.1: Python script of fitting example

```
 Chi2:1.02169224e-02    chi2.NCall:155  grad.NCall:0,0,0 (neval, ngrad, total
----------------------------------------------------
  MinimizerType          : Minuit2
  MinimizerAlgorithm     : Migrad
--- Options ------------------------------------------
  Strategy               : 1
  Tolerance              : 1.00000000e-02
  MaxFunctionCalls       : 10000
  MaxIterations          : 10000
  Precision              : -1.00000000e+00
  ErrorDefinition        : 1.00000000e+00 (1-chi2, 0.5 likelihood)
  ExtraOptions           : 0
--- Status -------------------------------------------
  Status                 : 0 'OK, valid minimum'
  IsValidError           : 0 'No detailed error validation'
  CovMatrixStatus        : 3 'full accurate'
  NCalls                 : 155
  MinValue               : 1.02162327e-02
  Edm                    : 5.29113396e-08
--- Variables ----------------------------------------
  NumberOfVariables      : 4 (free), 4 (total)
  Errors                 : yes, see below
  Npar  Name                            Value         Error
GlobalCC
  0    *FormFactorCylinder/height      4.999918e+00  2.127799e-01
8.620908e-01
  1    *FormFactorCylinder/radius      4.999933e+00  9.159691e-02
8.666066e-01
  2    *FormFactorPrism3/height        5.000522e+00  4.802199e-01
8.595439e-01
  3    *FormFactorPrism3/half_side     5.000185e+00  2.542907e-01
8.687376e-01
--- Correlations-------------------------------------
      1.000000e+00  -2.045230e-01 -8.420057e-01 9.493872e-02
      -2.045230e-01 1.000000e+00  2.158473e-01  -8.497385e-01
      -8.420057e-01 2.158473e-01  1.000000e+00  -2.047101e-01
      9.493872e-02  -8.497385e-01 -2.047101e-01 1.000000e+00
```

Listing 5.2: Output of fit using Python script 5.1

The displayed output starts by giving the values of **Chi2**, **chi2NCall** (number of calls), **grad.NCall** (for gradient evaluation). The expression of χ^2 is

$$\sum_{\text{nb fitting parameters}} \Big(\frac{\text{weight}}{\text{total weight}}\Big)^2 \Big(\sum \frac{(\text{sim}-\text{ref})^2}{\text{number deg. freedom}}\Big),$$

TO CHECK where the intensity has been normalized with respect to ... and weight is equal to 1 by default.

Then a **Description of minimizer** used for this particular procedure is given with its name and the associated algorithm.

Options:

- `Strategy`: `Minuit2` strategy, equal to 1 by default (see [**?**] page 5),
- `Tolerance` = required tolerance on the function value at the minimum,
- `MaxFunctionCalls` = maximum number of function calls above which the calculation will stop,
- `MaxIterations` = maximum number of iterations,
- `Precision` = precision of minimizer in the evaluation of the objective function (a negative value corresponds to letting the minimizer choose its default one),
- `ErrorDefinition` returns the statistical scale used for calculate the error. It is typically 1 for Chi2 and 0.5 for likelihood minimization. It is equal to 1 by default,
- `ExtraOptions` returns 0 if no other option have been implemented.

Status:

- `Status` =

  0: `'OK, valid minimum'`,

  1: `'Didn't converge, covariance was made pos defined'`,

  2: `'Didn't converge, Hesse is invalid'`,

  3: `'Didn't converge, Edm is above max'`,

  4: `'Didn't converge, Reached call limit'`,

  5: `'Didn't converge, Any other failure'`.

- `IsValidError` returns true if Minimizer has performed a detailed error validation (*e.g.* run Hesse method for `Minuit`).

  0: `'No detailed error validation'`,

  1: `'Performed detailed error validation'`.

- `CovMatrixStatus`

  -1: `'not available (inversion failed or Hesse failed)'`,

  0: `'available but not positive defined'`,

  1: `'covariance only approximate'`,

  2: `'full matrix but forced pos def'` (`pos def` stands for positive definite),

  3: `'full accurate'`.

- `NCalls`: number of function calls to reach the minimum,
- `MinValue` returns minimum function value,
- `Edm` returns expected vertical distance to the minimum. (`Edm` stands for "expected distance to the minimum")

Variables:
This part reports the list of fitting parameters with their names, the values determined at the end of the minimization process, the errors...

- `NumberOfVariables`: number of free and total variables,
- `Errors` = `yes, see below` or `no access`.
  If `Errors = yes, see below`, an array is displayed. The number of lines is equal to the number `Npar` of fitting parameters `Name`, the `Value` is the one reached at the minimum, the `Error` corresponds to the errors at this minimum. `GlobalCC` returns global correlation coefficient for parameter i. It is comprised between zero and one.

Correlations displays the correlation matrix. Each coefficient is defined as $\text{CovMat}(i,j)/\sqrt{\text{CovMat}(i,i)\text{CovMat}(j,j)}$, where $\text{CovMat}(a,b)$ is the element of the covariant matrix at line a and column b. And it is comprised between -1 and 1.

Remark: Depending on the selected minimizer and the algorithm and if a local minimum has been found, the output might differ as, for example, the covariant matrix is not available for some minimizers.

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 5.1: List of fitting minimizers implemented in `BornAgain`.