BornAgain

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

User Manual
version 0.1.1
October 11, 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)
- 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 examples

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

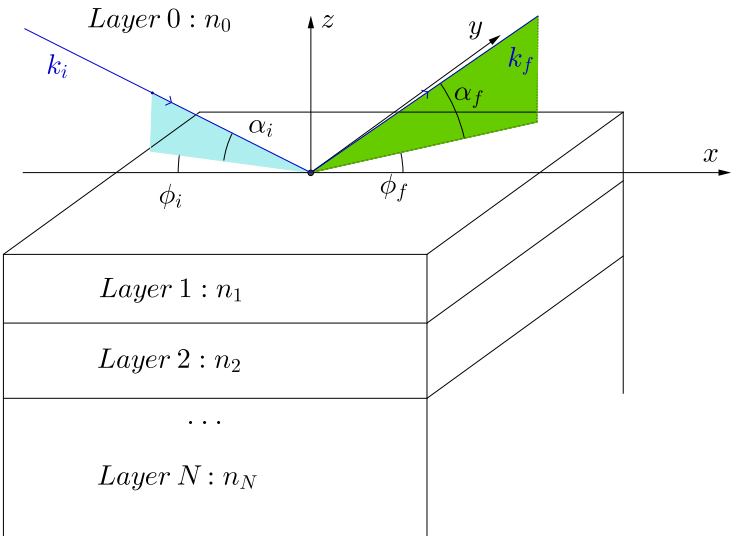

Figure 3.1: Representation of the scattering geometry. n_j is the refractive index of layer j and α_i and ϕ_i are the incident angle of the wave propagating. α_f is the exit angle with respect to the sample's surface and ϕ_f is the scattering angle with respect to the scattering plane.

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. 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 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.

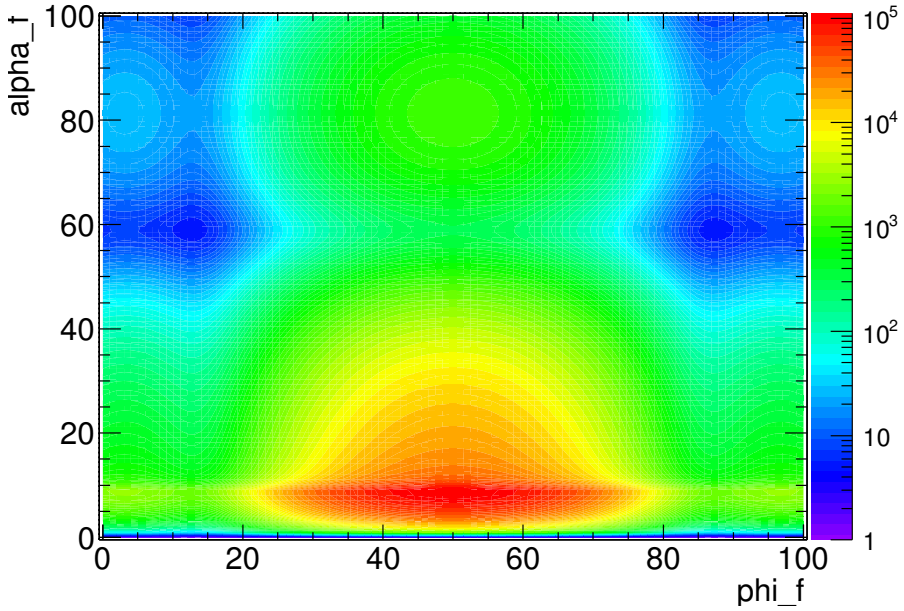

Figure 3.2: Figure of example 1: Simulated grazing-incidence small-angle X-ray scattering from a mixture of cylindrical and prismatic nanoparticles without any interference, deposited on top of a substrate. The input beam is characterized by a wavelength λ of 1 Å and incident angles $\alpha_i = 0.2°$,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

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. Details of the implementation in `BornAgain` are given in Section 4.2. Section 4.3 contains Python fitting example with detailed explanations of every fitting step.

4.1 Basic concept of data fitting.

4.1.1 Terminology.

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

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. 4.1 shows general work flow of fitting procedure. Before running the fitting the

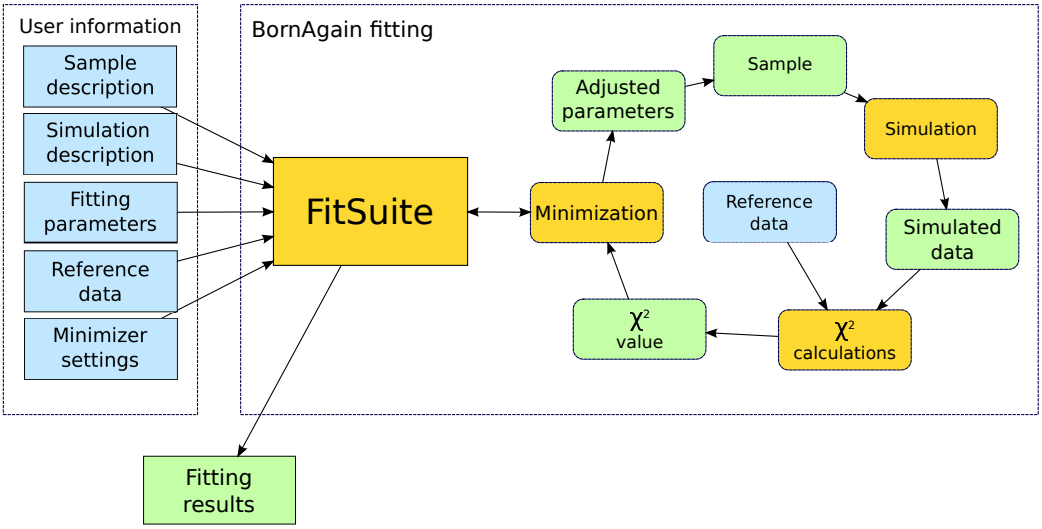

Figure 4.1: Fitting work flow.

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

- Preparing sample description.
- Preparing simulation description (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 aa 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.

The control is returned to the user

- 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

Some technical details of `FitSuite` class implementation 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.

4.3 Simple fitting python example.

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.

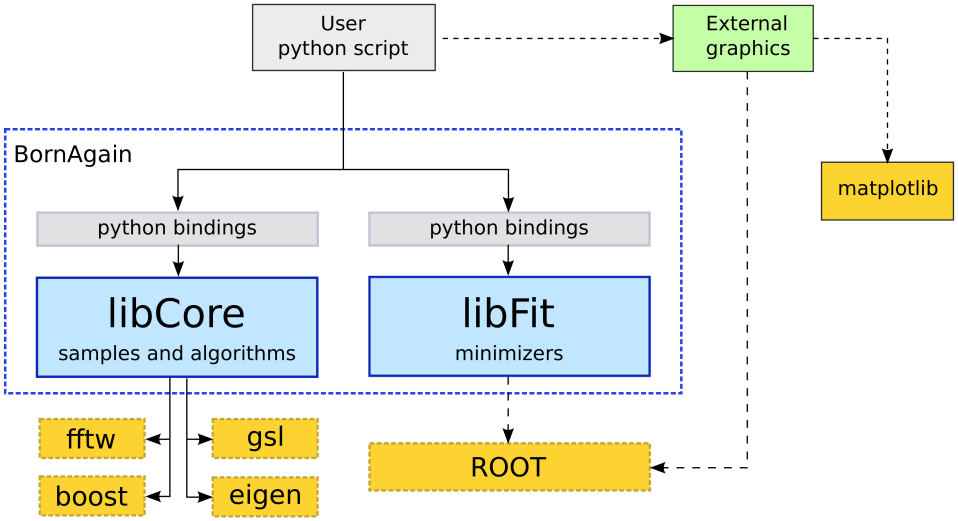

Figure 5.1: Structure of `BornAgain` libraries.

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.

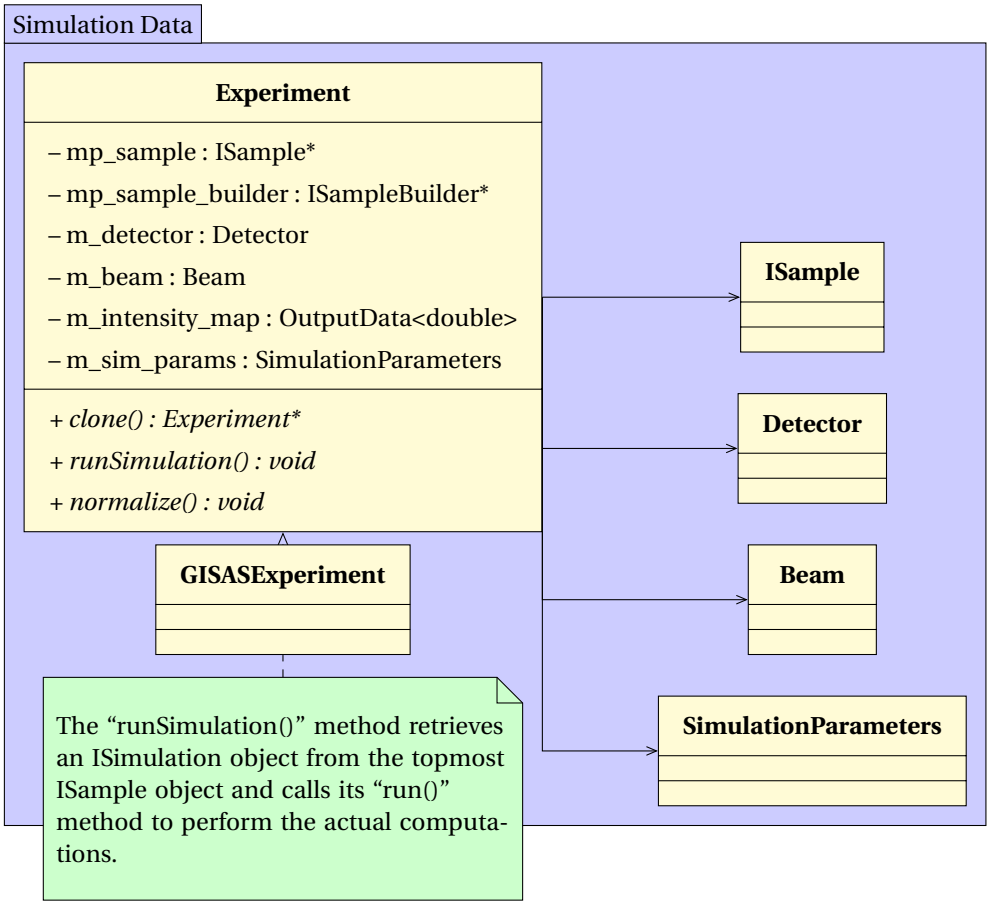

Figure 5.2: The Experiment class as a container for sample, beam, detector and simulation parameters.

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.

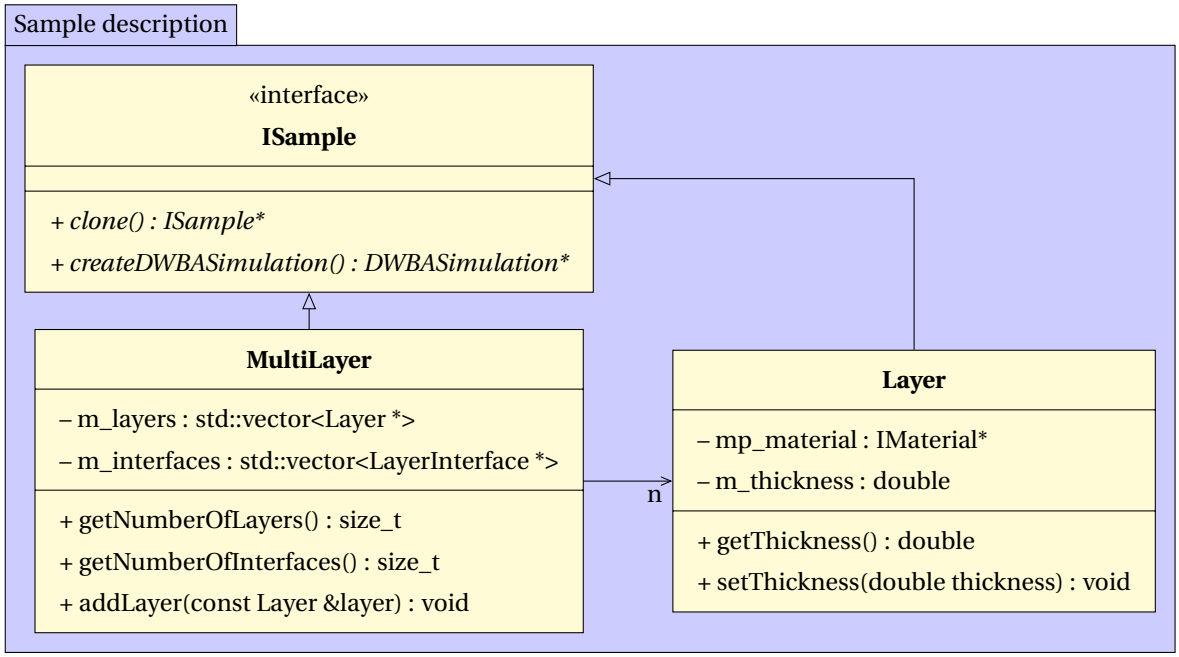

Figure 5.3: The ISample interface

5.1.3 The FitSuite class hierarchy