BornAgain - simulating and fitting X-ray and neutron scattering at grazing incidence.

User Guide
version 0.1

Scientific Computing Group at FRM-II

July 26, 2013

Contents

Listings

List of Tables

Chapter 1

Examples

1.1 General methodology

A simulation of GISAXS using `BornAgain` platform can be decomposed into the following points:

- Definition of the materials by specifying their names and their refractive indices,
- Definition of particles: shapes, sizes, refractive indices of the constituting material, interference functions,
- Definition of the layers: thicknesses, roughnesses, associations with the previously defined materials,
- Inclusion of the particles in layers: **density or proportion**, positions, orientations,
- Assembling the sample: generation of a multilayered system,
- Specifying the input beam and the output detector's characteristics,
- Running the simulation,
- Saving the data.

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

1.2 Conventions

1.2.1 Geometry of the sample

The geometry used to describe the sample is shown in Fig. 1.1. 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 by two angles α_0, ϕ_0 and α_f, ϕ_f respectively. Then for each other layer $j = 1, ..., N-1$, the incident angles α_j and ϕ_j are defined with respect to the bottom of the layer. Our choice of orientation for the angles α_j is so that α_0 and α_f shown in fig. 1.1 are positive.

The layers are defined by their thicknesses (parallel to the z-direction), their possible roughnesses (equal to 0 by default) and the refractive index of the material. 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`.

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 refractive index of 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. 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.

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. But for all the other layers, it is the top of the layer which corresponds to `depth`=0.

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.

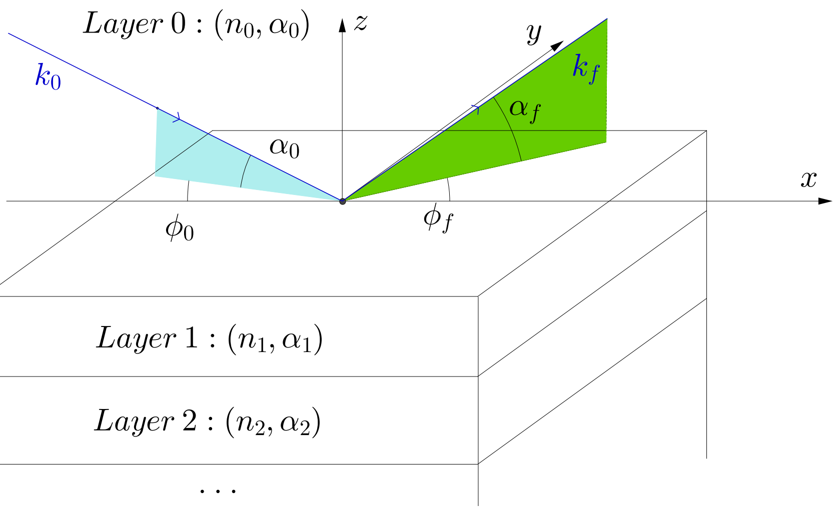
$Layer\ 0 : (n_0, \alpha_0)$
z
y
k_0
k_f
α_f
α_0
x
ϕ_0
ϕ_f
$Layer\ 1 : (n_1, \alpha_1)$
$Layer\ 2 : (n_2, \alpha_2)$
...

Figure 1.1: Representation of the scattering geometry for multilayer specular reflectivity. n_j is the refractive index of layer j and α_j is the incident angle of the wave propagating in layer j and incident on layer $j+1$. α_f is the exit angle with respect to the sample's surface and ϕ_f is the scattering angle with respect to the scattering plane.

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

1.2.2 Units

By default angles are expressed in radians and 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*Units::micrometer` in C++.

1.2.3 Programs

✎ Programming The examples presented in the next paragraphs are written in C++ or Python. For tutorials about these programming languages, the users are referred to [1] and [2] respectively.

Note about the version of C++ and Python to run the examples.

Where can the following examples be found?

What is the command to run the examples?

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

In this example, using Python language, 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 1.1).

We start by importing different functions from external modules (lines 1-7). For example, line 3 imports NumPy, which is a fundamental package for scientific computing with Python (`http://www.numpy.org/`). In particular, line 7 imports the features of `BornAgain` software.

Finally line 9 marks the beginning of the function to define and run the simulation.

```
1  import sys
2  import os
3  import numpy
4
5  sys.path.append(os.path.abspath(os.path.join(os.path.split(__file__)[0],
       '..', '..', '..', 'lib')))
6
7  from libBornAgainCore import *
8
9  def RunSimulation():
```

First step: Defining materials

```
10 #  defining materials
11 mAmbience = MaterialManager.getHomogeneousMaterial("Air", 0.0, 0.0 )
```

```
12 mSubstrate = MaterialManager.getHomogeneousMaterial("Substrate", 6e-6, 2
       e-8)
```

Lines 11 and 12 define two different materials using function `getHomogeneousMaterial` from class `MaterialManager`. The general syntax is the following

```
Interface 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`. `Interface material name` is later used when referring to this particular material. The two defined materials in this example are `Air` with a refractive index of 1 (`delta = beta =0`) and a `Substrate` associated with a complex refractive index equal to $1-6\times10^{-6}+i2\times10^{-8}$.

Remark: there is no condition on the choice of `name`.

Second step: Defining the particles

```
13 # collection of particles
14 n_particle = complex(6e-4, 2e-8)
15 cylinder_ff = FormFactorCylinder(5*nanometer, 5*nanometer)
16 cylinder = Particle(n_particle, cylinder_ff)
17 prism_ff = FormFactorPrism3(5*nanometer, 5*nanometer)
18 prism = Particle(n_particle, prism_ff)
```

We implement two different shapes of particles: cylinders and prisms (*i.e.* elongated particles with a constant equilateral triangular cross section).
All particles implemented in `BornAgain` are defined by their form factors, their sizes and the refractive index of the material they are made of. Here, for the cylindrical particle, we input its radius and its height. For the prism, the possible inputs are the length of one side of its equilateral triangular base and its height.

In line 14, we define the complex refractive index associated with both particle shapes: `n=` $1-6\times10^{-4}+i2\times10^{-8}$.

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 15). Then we associate this shape with the refractive index of the constituting material as in line 16.

The same procedure has been applied for the prism in lines 17 and 18 respectively.

Third step: Characterizing the layers and assembling the sample

Particle decoration

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

The process of defining the positions and densities of particles in our sample is called “particle decoration”. We use the function `ParticleDecoration()` (line 19) and the associated `addParticle` for each particle shape (lines 20, 21). The general syntax is

```
particledecoration.addParticle(particle_name, depth, abundance)
```

where `particle_name` is the name used to define the particles (lines 16 and 18), `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 fig. 1.1 is pointing upwards. But for all the other layers, it is the top of the layer which corresponds to `depth`=0.

Finally lines 22 and 23 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 20 and 21.

Multilayer

```
24 # air layer with particles and substrate form multi layer
25 air_layer = Layer(mAmbience)
26 air_layer_decorator = LayerDecorator(air_layer, particle_decoration)
27 substrate_layer = Layer(mSubstrate, 0)
28 multi_layer = MultiLayer()
29 multi_layer.addLayer(air_layer_decorator)
30 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 25, we use the previously defined `mAmbience` (="air" material) (line 11). The command written in line 26 shows that this layer is decorated by adding the particles using the function `particledecoration` defined in lines 19-23. Note that the `depth` is referenced to the bottom of the top layer (negative values would correspond to particles floating above layer 1 as the vertical axis is pointing upwards). The substrate layer only contains the substrate material (line 27).

There are different possible syntaxes to define a layer. As shown in lines 25 and 27, we can use `Layer(Interface material name,thickness)` or `Layer(Interface 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 28): we start with the air layer decorated with the particles (line 29), which is the layer at the top and end with the bottom layer, which is the substrate (line 30).

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

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

The first stage is to define the `Simulation()` object (line 32). Then we define the detector (line 34) and beam parameters (line 35), which are associated with the sample previously defined (line 36). Finally we run the simulation (line 37). Those functions are part of the Simulation class. The different incident and exit angles are shown in Fig. 1.1.

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 points in the range of variations of angle 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, which is the in-plane direction of the scattered beam (measured with respect to the x-axis),
`n_alpha=100` is the number of points in the range of variations of the exit angle α_f measured from the x,y-plane in the z-direction,
`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 `simulation.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). Note that in Fig. 1.1 $\alpha_i = \alpha_0$ and $\phi_i = \phi_0$.

Remark: Note that, except for `isgisaxs_style`, there are no default values implemented for the parameters of the beam and detector.

Line 37 shows the command to run the simulation using the previously defined setup.

Fifth step: Saving the data

```
38 # retrieving intensity data
39 return GetOutputData(simulation) %arr = GetOutputData(simulation)
```

In line 39 we record 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 1.2 shows the two-dimensional contourplot of the intensity as a function of α_f and ϕ_f.

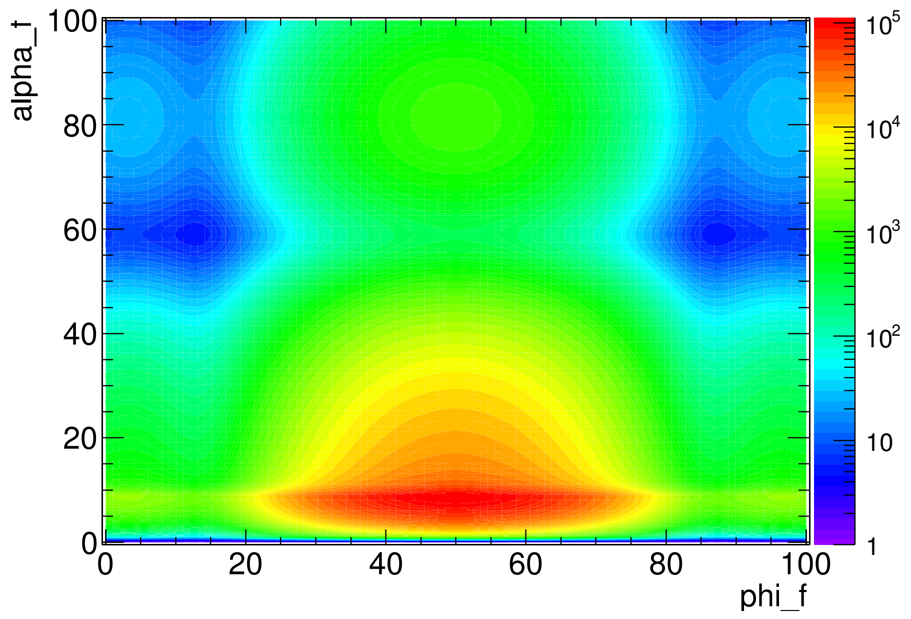

Figure 1.2: Figure of example 1: Simulated grazing-incidence small-angle X-ray scattering from a mixture of cylindrical and prismatic nanoparticles without any interference, deposited on top of a substrate. The input beam is characterized by a wavelength λ of 1 Å and incident angles $\alpha_i = 0.2°$, $\phi_i = 0°$. 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 $1-6\times10^{-4}+i2\times10^{-8}$. For the substrate it is equal to $1-6\times10^{-6}+i2\times10^{-8}$. The colorscale is associated with the output intensity in arbitrary units.

```python
import sys
import os
import 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)
    # collection of particles
    n_particle = complex(6e-4, 2e-8)
    cylinder_ff = FormFactorCylinder(5*nanometer, 5*nanometer)
    cylinder = Particle(n_particle, cylinder_ff)
    prism_ff = FormFactorPrism3(5*nanometer, 5*nanometer)
    prism = Particle(n_particle, 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_decorator = LayerDecorator(air_layer, particle_decoration)
    substrate_layer = Layer(mSubstrate, 0)
    multi_layer = MultiLayer()
    multi_layer.addLayer(air_layer_decorator)
    multi_layer.addLayer(substrate_layer)

    # 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 1.1: Python script of example 1

1.4 Example 2

Bibliography

[1] Kyle Loudon. *C++ pocket reference.* O'Reilly media, 2008.

[2] Mark Lutz. *Python pocket reference.* O'Reilly media, fourth edition edition, 2009.

[3] Minuit user's guide. `http://seal.web.cern.ch/seal/documents/minuit/mnusersguide.pdf`.