BornAgain: simulating X-ray and neutron scattering at grazing incidence

Céline Durniak, Walter Van Herck, Gennady Pospelov, Joachim Wuttke

Scientific Computing Group at FRM-II
Jülich Center for Neutron Science

Emerging themes in Analysis of Grazing Incidence Small Angle Scattering Data
1st and 2nd July, Cosener’s House, Abingdon
Outline

- Motivation
- Architecture
- Usage examples
- Infrastructure for code development
- Future plans
Motivation

Heinz Maier-Leibnitz Zentrum (MLZ)
FRM-II neutron source in Garching bei Munich

Scientific computing at FRM-II
- Support of instruments with focus on simulation and data analysis

HDRI
- High Data Rate Processing and Analysis Initiative

From local (MLZ) and more broader needs (HDRI)
- Start community project for different user experiments (neutron and x-ray grazing incidence)
- More generally structured than existing software
- IsGISAXS as reference software
IsGISAXS as a starting point:

- Successful software which is a de facto standard in the user community

**IsGISAXS**: a program for grazing-incidence small-angle X-ray scattering analysis of supported islands

R Lazzari - Journal of Applied Crystallography, 2002 - scripts.iucr.org

This paper describes a Fortran program, IsGISAXS, for the simulation and analysis of grazing-incidence small-angle X-ray scattering (GISAXS) of islands supported on a substrate. As is usual in small-angle scattering of particles, the scattering cross section is ...

Cited by 257 Related articles All 7 versions Cite

- Simulation in DWBA
- FORTRAN 90, 13k lines of code
- Windows (with GUI), Linux (without GUI)
- No longer actively supported
New software platform or extend existing?

Extending IsGISAXS
- code reuse

From scratch C++
- developers background
- object oriented approach in sample description
Reproduce functionality of IsGISAXS
  o IsGISAXS examples as milestones along the way

Extend it with most demanded features
  o Multilayers, particle assemblies, polarized neutrons and magnetic domains

Follow rules
  o Decouple physical modelling, data, GUI
  o Depend on open source and actively maintained libraries
  o Produce open source code
  o Be platform independent

Build user community and follow their demands
The name of the software, BornAgain, indicates the central role of the Distorted Wave Born Approximation

- generic frame for modeling and fitting multi layer samples at grazing incidence geometry

**Supported sample structures**

- Multilayer
- Interface roughness (also correlated)
- Multiple nanoparticles (shapes, densities)
- Interference functions
- Nanoparticles assemblies

**Supports X-rays and non-polarized neutrons**

**Current status – beta**
Transition to production release with user help
Main features:
- Programming language C++
- Core library consist of 25k lines of code
- Python bindings with boost-python
- Mac/Linux, Windows soon
Basic software architecture

External dependencies:
- Well established libraries: Boost, fftw3, Eigen, GSL
- ROOT from High Energy Physics community for fitting
Working with BornAgain.

- Running user C++ program
Working with BornAgain.

- Running user Python script

Basic software architecture

User Python script

- BornAgain (Python bindings)

  - libCore
    - Samples and algorithms

  - libFit
    - Interface to minimizers

- External graphics

  - Matplotlib

- GSL
- Boost
- Eigen
- (Py)ROOT
Example #1: simulation of cylindrical nano particles with interference function on top of substrate using python script
Usage example: simulation with python script

```python
from libBornAgainCore import *

# defining materials
ambience = HomogeneousMaterial("Air", 1.0, 0.0)
substrate = HomogeneousMaterial("Substrate", 1.0-6e-6, 2e-8)

# two empty layers
air_layer = Layer(ambience)
substrate_layer = Layer(substrate)

# creating particles
cylinder_ff = FormFactorPyramid(5*nm, 5*nm, 54.73*deg)
n_particle = complex(1.0-5e-5, 2e-8)
cylinder = Particle(n_particle, cylinder_ff)

# additional properties: transformation
angle_around_z = 45.*degree
transform = RotateZ3D(angle_around_z)

# additional properties: interference function
interference = InterferenceFunction1DParaCrystal(20*nm, 7*nm, 1e7*nm)

# decorating air layer with particles
particle_decoration = ParticleDecoration()
particle_decoration.addParticle(cylinder, transform)
particle_decoration.addInterferenceFunction(interference)
decorated_layer = LayerDecorator(air_layer, particle_decoration)

# creating multilayer
multi_layer = MultiLayer()
multi_layer.addLayer(decorated_layer)
multi_layer.addLayer(substrate_layer)

# simulating
simulation = Simulation()
simulation.setDetector(100, 0.0*deg, 2.0*deg, 100, 0.0*deg, 2.0*deg)
simulation.setBeamParameters(1.0*angstrom, -0.2*deg, 0.0*deg)
simulation.setSample(multi_layer)
simulation.runSimulation()

# retrieving data
arr = GetOutputData(simulation)
```
from libBornAgainCore import *

# defining materials
ambience = HomogeneousMaterial("Air", 1.0, 0.0)
substrate = HomogeneousMaterial("Substrate", 1.0-6e-6, 2e-8)

# two empty layers
air_layer = Layer(ambience)
substrate_layer = Layer(substrate)

# creating particles
cylinder_ff = FormFactorPyramid(5*nm, 5*nm, 54.73*deg)
n_particle = complex(1.0-5e-5, 2e-8)
cylinder = Particle(n_particle, cylinder_ff)

# additional properties: transformation
angle_around_z = 45.*degree
transform = RotateZ3D(angle_around_z)

# additional properties: interference function
interference = InterferenceFunction1DParaCrystal(20*nm, 7*nm, 1e7*nm)

# decorating air layer with particles
particle_decoration = ParticleDecoration()
particle_decoration.addParticle(cylinder, transform)
particle_decoration.addInterferenceFunction(interference)
decorated_layer = LayerDecorator(air_layer, particle_decoration)

# creating multilayer
multi_layer = MultiLayer()
multi_layer.addLayer(decorated_layer)
multi_layer.addLayer(substrate_layer)

# simulating
simulation = Simulation()
simulation.setDetector(100, 0.0*deg, 2.0*deg, 100, 0.0*deg, 2.0*deg)
simulation.setBeamParameters(1.0*angstrom, -0.2*deg, 0.0*deg)
simulation.setSample(multi_layer)
simulation.runSimulation()

#retrieving data
arr = GetOutputData(simulation)
Usage example: simulation with python script

from libBornAgainCore import *

# defining materials
ambience = HomogeneousMaterial("Air", 1.0, 0.0)
substrate = HomogeneousMaterial("Substrate", 1.0-6e-6, 2e-8)

# two empty layers
air_layer = Layer(ambience)
substrate_layer = Layer(substrate)

cylinder_ff = FormFactorPyramid(5*nm, 5*nm, 54.73*deg)
n_particle = complex(1.0-5e-5, 2e-8)
cylinder = Particle(n_particle, cylinder_ff)

# additional properties: transformation
angle_around_z = 45.*degree
transform = RotateZ3D(angle_around_z)

# additional properties: interference function
interference = InterferenceFunction1DParaCrystal(20*nm, 7*nm, 1e7*nm)

# decorating air layer with particles
particle_decoration = ParticleDecoration()
particle_decoration.addParticle(cylinder, transform)
particle_decoration.addInterferenceFunction(interference)
decorated_layer = LayerDecorator(air_layer, particle_decoration)

# creating multilayer
multi_layer = MultiLayer()
multi_layer.addLayer(decorated_layer)
multi_layer.addLayer(substrate_layer)

# simulating
simulation = Simulation()
simulation.setDetector(100, 0.0*deg, 2.0*deg, 100, 0.0*deg, 2.0*deg)
simulation.setBeamParameters(1.0*angstrom, -0.2*deg, 0.0*deg)
simulation.setSample(multi_layer)
simulation.runSimulation()

# retrieving data
arr = GetOutputData(simulation)
Nano particle is defined by the refractive index and form factor.

Additional transformation can be defined for orientation and position of the particle.

```
from libBornAgainCore import *
# defining materials
ambi = HomogeneousMaterial("Air", 1.0, 0.0 )
substrate = HomogeneousMaterial("Substrate", 1.0-6e-6, 2e-8 )
# two empty layers
air_layer = Layer(ambi)
substrate_layer = Layer(substrate)
# creating particles
cylinder_ff = FormFactorPyramid(5*nm, 5*nm, 54.73*deg)
n_particle = complex(1.0-5e-5, 2e-8)
cylinder = Particle(n_particle, cylinder_ff)
# additional properties: transformation
angle_around_z = 45.*degree
transform = RotateZ3D(angle_around_z)
# additional properties: interference function
interference = InterferenceFunction1DParaCrystal(20*nm, 7*nm, 1e7*nm)
# decorating air layer with particles
particle_decoration = ParticleDecoration()
particle_decoration.addParticle(cylinder, transform)
particle_decoration.addInterferenceFunction(interference)
decorated_layer = LayerDecorator(air_layer, particle_decoration)
# creating multilayer
multi_layer = Multilayer()
multi_layer.addLayer(decorated_layer)
multi_layer.addLayer(substrate_layer)
# simulating
simulation = Simulation()
simulation.setDetector(100, 0.0*deg, 2.0*deg, 100, 0.0*deg, 2.0*deg)
simulation.setBeamParameters(1.0*angstrom, -0.2*deg, 0.0*deg)
simulation.setSample(multi_layer)
simulation.runSimulation()
#retrieving data
arr = GetOutputData(simulation)
```
InterferenceFunction1DParaCrystal defines interference function of regular lattice with long range order gradually destroyed.

```python
from libBornAgainCore import *

# defining materials
ambience = HomogeneousMaterial("Air", 1.0, 0.0)
substrate = HomogeneousMaterial("Substrate", 1.0-6e-6, 2e-8)

# two empty layers
air_layer = Layer(ambience)
substrate_layer = Layer(substrate)

# creating particles
cylinder_ff = FormFactorPyramid(5*nm, 5*nm, 54.73*deg)
n_particle = complex(1.0-5e-5, 2e-8)
cylinder = Particle(n_particle, cylinder_ff)

# additional properties: transformation
angle_around_z = 45.*degree
transform = RotateZ3D(angle_around_z)

# additional properties: interference function
interference = InterferenceFunction1DParaCrystal(20*nm, 7*nm, 1e7*nm)

# decorating air layer with particles
particle_decoration = ParticleDecoration()
particle_decoration.addParticle(cylinder, transform)
particle_decoration.addInterferenceFunction(interference)
decorated_layer = LayerDecorator(air_layer, particle_decoration)

# creating multilayer
multi_layer = MultiLayer()
multi_layer.addLayer(decorated_layer)
multi_layer.addLayer(substrate_layer)

# simulating
simulation = Simulation()
simulation.setDetector(100, 0.0*deg, 2.0*deg, 100, 0.0*deg, 2.0*deg)
simulation.setBeamParameters(1.0*angstrom, -0.2*deg, 0.0*deg)
simulation.setSample(multi_layer)
simulation.runSimulation()

# retrieving data
arr = GetOutputData(simulation)
```
Usage example: simulation with python script

```python
from libBornAgainCore import *

# defining materials
ambience = HomogeneousMaterial("Air", 1.0, 0.0)
substrate = HomogeneousMaterial("Substrate", 1.0-6e-6, 2e-8)

# two empty layers
air_layer = Layer(ambience)
substrate_layer = Layer(substrate)

# creating particles
cylinder_ff = FormFactorPyramid(5*nm, 5*nm, 54.73*deg)
n_particle = complex(1.0-5e-5, 2e-8)
cylinder = Particle(n_particle, cylinder_ff)

# additional properties: transformation
angle_around_z = 45.*degree
transform = RotateZ3D(angle_around_z)

# additional properties: interference function
interference = InterferenceFunction1DParaCrystal(20*nm, 7*nm, 1e7*nm)

# decorating air layer with particles
particle-decoration = ParticleDecoration()
particle-decoration.addParticle(cylinder, transform)
particle-decoration.addInterferenceFunction(interference)
decorated_layer = LayerDecorator(air_layer, particle-decoration)

# creating multilayer
multi_layer = MultiLayer()
multi_layer.addLayer(decorated_layer)
multi_layer.addLayer(substrate_layer)

# simulating
simulation = Simulation()
simulation.setDetector(100, 0.0*deg, 2.0*deg, 100, 0.0*deg, 2.0*deg)
simulation.setBeamParameters(1.0*angstrom, -0.2*deg, 0.0*deg)
simulation.setSample(multi_layer)
simulation.runSimulation()

#retrieving data
arr = GetOutputData(simulation)
```

**InterferenceFunction1DParaCrystal**
defines interference function of regular lattice with long range order gradually destroyed

**ParticleDecoration**
Holds information about particle types, positions and interference
Usage example: simulation with python script

```
from libBornAgainCore import *

# defining materials
ambience = HomogeneousMaterial("Air", 1.0, 0.0)
substrate = HomogeneousMaterial("Substrate", 1.0-6e-6, 2e-8)

# two empty layers
air_layer = Layer(ambience)
substrate_layer = Layer(substrate)

# creating particles
cylinder_ff = FormFactorPyramid(5*nm, 5*nm, 54.73*deg)
n_particle = complex(1.0-5e-5, 2e-8)
cylinder = Particle(n_particle, cylinder_ff)

# additional properties: transformation
angle_around_z = 45.*degree
transform = RotateZ3D(angle_around_z)

# additional properties: interference function
interference = InterferenceFunction1DParaCrystal(20*nm, 7*nm, 1e7*nm)

# decorating air layer with particles
particle_decoration = ParticleDecoration()
particle_decoration.addParticle(cylinder, transform)
particle_decoration.addInterferenceFunction(interference)
decorated_layer = LayerDecorator(air_layer, particle_decoration)

# creating multilayer
multi_layer = MultiLayer()
multi_layer.addLayer(decorated_layer)
multi_layer.addLayer(substrate_layer)

# simulating
simulation = Simulation()
simulation.setDetector(100, 0.0*deg, 2.0*deg, 100, 0.0*deg, 2.0*deg)
simulation.setBeamParameters(1.0*angstrom, -0.2*deg, 0.0*deg)
simulation.setSample(multi_layer)
simulation.runSimulation()

# retrieving data
arr = GetOutputData(simulation)
```

**InterferenceFunction1DParaCrystal**
Defines interference function of regular lattice with long range order gradually destroyed.

**ParticleDecoration**
Holds information about particle types, positions and interference.

**LayerDecorator**
Decorates air layer with particle decoration.
**Usage example: simulation with python script**

```
from libBornAgainCore import *

# defining materials
ambience = HomogeneousMaterial("Air", 1.0, 0.0)
substrate = HomogeneousMaterial("Substrate", 1.0e-6, 2e-8)

# two empty layers
air_layer = Layer(ambience)
substrate_layer = Layer(substrate)

# creating particles
cylinder_ff = FormFactorPyramid(5*nm, 5*nm, 54.73*deg)
n_particle = complex(1.0-5e-5, 2e-8)
cylinder = Particle(n_particle, cylinder_ff)

# additional properties: transformation
angle_around_z = 45.*degree
transform = RotateZ3D(angle_around_z)

# additional properties: interference function
interference = InterferenceFunction1DParaCrystal(20*nm, 7*nm, 1e7*nm)

# decorating air layer with particles
particle_decoration = ParticleDecoration()
particle_decoration.addParticle(cylinder, transform)
particle_decoration.addInterferenceFunction(interference)
decorated_layer = LayerDecorator(air_layer, particle_decoration)

# creating multilayer
multi_layer = MultiLayer()
multi_layer.addLayer(decorated_layer)
multi_layer.addLayer(substrate_layer)

# simulating
simulation = Simulation()
simulation.setDetector(100, 0.0*deg, 2.0*deg, 100, 0.0*deg, 2.0*deg)
simulation.setBeamParameters(1.0*angstrom, -0.2*deg, 0.0*deg)
simulation.setSample(multi_layer)
simulation.runSimulation()

#retrieving data
arr = GetOutputData(simulation)
```

**MultiLayer**

owns decorated air layer and substrate layer

---

Mul/Layer owns decorated air layer and substrate layer
Usage example: simulation with python script

**MultiLayer**

owns decorated air layer and substrate layer

```
from libBornAgainCore import *

# defining materials
ambience = HomogeneousMaterial("Air", 1.0, 0.0)
substrate = HomogeneousMaterial("Substrate", 1.0-6e-6, 2e-8)

# two empty layers
air_layer = Layer(ambience)
substrate_layer = Layer(substrate)

# creating particles
cylinder_ff = FormFactorPyramid(5*nm, 5*nm, 54.73*deg)
n_particle = complex(1.0-5e-5, 2e-8)
cylinder = Particle(n_particle, cylinder_ff)

# additional properties: transformation
angle_around_z = 45.*degree
transform = RotateZ3D(angle_around_z)

# additional properties: interference function
interference = InterferenceFunction1DParaCrystal(20*nm, 7*nm, 1e7*nm)

# decorating air layer with particles
particle_decoration = ParticleDecoration()
particle_decoration.addParticle(cylinder, transform)
particle_decoration.addInterferenceFunction(interference)
decorated_layer = LayerDecorator(air_layer, particle_decoration)

# creating multilayer
multi_layer = MultiLayer()
multi_layer.addLayer(decorated_layer)
multi_layer.addLayer(substrate_layer)

# simulating
simulation = Simulation()
simulation.setDetector(100, 0.0*deg, 2.0*deg, 100, 0.0*deg, 2.0*deg)
simulation.setBeamParameters(1.0*angstrom, -0.2*deg, 0.0*deg)
simulation.setSample(multi_layer)
simulation.runSimulation()

# retrieving data
arr = GetOutputData(simulation)
```

**Simulation**

Holds beam and detector parameters, accept multi layer as an input and perform calculations
OutputData

Simulated intensity as a function of outgoing $\alpha_f$, $\phi_f$ angles can be retrieved as NumPy array

```python
from libBornAgainCore import *

# defining materials
ambience = HomogeneousMaterial("Air", 1.0, 0.0)
substrate = HomogeneousMaterial("Substrate", 1.0-6e-6, 2e-8)

# two empty layers
air_layer = Layer(ambience)
substrate_layer = Layer(substrate)

# creating particles
cylinder_ff = FormFactorPyramid(5*nm, 5*nm, 54.73*deg)
n_particle = complex(1.0-5e-5, 2e-8)
cylinder = Particle(n_particle, cylinder_ff)

# additional properties: transformation
angle_around_z = 45.*degree
transform = RotateZ3D(angle_around_z)

# additional properties: interference function
interference = InterferenceFunction1DParaCrystal(20*nm, 7*nm, 1e7*nm)

# decorating air layer with particles
particle_decodation = ParticleDecoration()
particle_decodation.addParticle(cylinder, transform)
particle_decodation.addInterferenceFunction(interference)
decorated_layer = LayerDecorator(air_layer, particle_decodation)

# creating multilayer
multi_layer = MultiLayer()
multi_layer.addLayer(decorated_layer)
multi_layer.addLayer(substrate_layer)

# simulating
simulation = Simulation()
simulation.setDetector(100, 0.0*deg, 2.0*deg, 100, 0.0*deg, 2.0*deg)
simulation.setBeamParameters(1.0*angstrom, -0.2*deg, 0.0*deg)
simulation.sampleSample(multi_layer)
simulation.runSimulation()

# retrieving data
arr = GetOutputData(simulation)
```
Example #2: fitting the data
Usage example: fitting

- Every number (or group of numbers) used in sample construction can be used as fit parameter.

### Minimization algorithms provided by ROOT

<table>
<thead>
<tr>
<th>Library</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minuit2</td>
<td>Migrad, Simplex, Fumili, Scan</td>
</tr>
<tr>
<td>GSL</td>
<td>Fletcher-Reeves Conjugate Gradient Algorithm</td>
</tr>
<tr>
<td></td>
<td>BFGS Conjugate Gradient Algorithm</td>
</tr>
<tr>
<td></td>
<td>Levenberg-Marquardt Algorithm</td>
</tr>
<tr>
<td></td>
<td>Simulated Annealing Algorithm</td>
</tr>
<tr>
<td>TMVA</td>
<td>Genetic Algorithm</td>
</tr>
</tbody>
</table>

```python
from libBornAgainCore import *
from libBornAgainFit import *

# creating minimizer
minimizer = MinimizerFactory.createMinimizer("Minuit2","Migrad")

# creating fitting engine
fitSuite = FitSuite()
fitSuite.setMinimizer(minimizer)

# creating fit parameters
fitSuite.addFitParameter("*/FormFactorCylinder/height", 4.0*nm, AttLimits.lowerLimited(0.01))
fitSuite.addFitParameter("*/FormFactorCylinder/radius", 4.0*nm, AttLimits.lowerLimited(0.01))

# setting simulation (from previous example) and real ASCII data
fitSuite.addSimulationAndRealData(simulation, data)

# running fit
fitSuite.runFit()
```
Example #3: mesocrystals
Usage example: mesocrystals

Grazing incidence small-angle x-ray scattering from a mesocrystalline system
- mesocrystals have cylindrical shape and size of ~1000 nm
- mesocrystal consist of an FCC lattice composed of 5 nm spherical particles

\[ S_{MC}(r) = S_{cyl}(r) \cdot \sum_{\{R_i\}} S_{NP}(r) \otimes \delta(r - R_i) \]
\[ F_{MC}(q) = F_{cyl}(q) \otimes \sum_{\{Q_i\}} F_{NP}(q) \cdot \delta(q - Q_i) \]
\[ = \sum_{\{Q_i\}} F_{cyl}(q - Q_i) \cdot F_{NP}(Q_i) \]
Usage example: mesocrystals

Experimental data compared to a BornAgain simulation

- 11 parameters fit
- 12 hours on six core (12 threads), 2000 iterations, 20 sec/iteration

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>beam_intensity</td>
<td>5.01e+12</td>
</tr>
<tr>
<td>lattice_length_a</td>
<td>6.21</td>
</tr>
<tr>
<td>lattice_length_c</td>
<td>6.57</td>
</tr>
<tr>
<td>nanoparticle_radius</td>
<td>4.70</td>
</tr>
<tr>
<td>sigma_nanoparticle_radius</td>
<td>0.37</td>
</tr>
<tr>
<td>meso_height</td>
<td>112.22</td>
</tr>
<tr>
<td>meso_radius</td>
<td>945.63</td>
</tr>
<tr>
<td>sigma_meso_height</td>
<td>1.00</td>
</tr>
<tr>
<td>sigma_meso_radius</td>
<td>1.61</td>
</tr>
<tr>
<td>sigma_lattice_length_a</td>
<td>1.16</td>
</tr>
<tr>
<td>surface_filling_ratio</td>
<td>0.17</td>
</tr>
<tr>
<td>roughness</td>
<td>28.72</td>
</tr>
</tbody>
</table>

experimental data
courtesy Elisabeth Josten et al

BornAgain simulation
in collaboration with Artur Glavic and Elisabeth Josten
Example #4: working with graphical user interface
Usage example: development of graphical user interface

Status – prototyping

- rely on Qt5 (Qt4) and ROOT libraries
- BornAgainCore library used as a plugin

In plans

- drag and drop sample editor
- interactive plots
- access to large collection of minimizers
- Python and C++ scripting
Development process

General practices

- More agile development
- Quality control
- Good documentation

Software tools

- Bug/issue tracking: Redmine
- Version control: Git
- Nightly build server: TeamCity
- Code documentation: Doxygen
- Code browser: OpenGrok
- Unit tests: Googletest
Development process: agile development

- workflow consists of sprint cycles every 3-4 weeks during which the team creates finished portions of the product.

Users

Developers

Backlog:
- Item 124
- Item 125
- Item 127
- ...

Sprint:
- Item 98
- Item 101
- Item 102
- ...

Release

One day I want to fly away...

Item 124
Learn to fly

Item 125
Fly away
Development process: agile development

Workflow is managed via Redmine management tool
- user request handling, bug tracking tool, issue tracking
Development process: quality assurance

Unit test control
- small test cases are executed after each compilation to ensure code stability

Functional tests
- higher level tests produce scattering plots which are automatically analyzed
- special tests compare BornAgain results with IsGISAXS results
Conclusion and plans

- **BornAgain** is a new software for simulating and fitting grazing incidence small angle scattering (x-rays and neutrons)
- Current status – beta
- It reproduces and enhances the functionality of the present reference software, IsGISAXS
- Everyone is welcome to try it out, when their experiment requires extension of IsGISAXS

Plans

- Usable GUI, Windows version (October-November, 2013)
- Neutron polarization and magnetic domains (April, 2014)

http://apps.jcns.fz-juelich.de/doku/sc/bornagain:start
Development process: quality assurance

Nightly build system
- based on TeamCity, one build configuration

Code profiling
- regular tracking of memory and CPU consumption
Development process: good documentation

Code documentation
- Doxygen generated, starting to write user manual

Theory and conventions
- described and stored in the repository parallel to the code development

Scattering on nanoparticles
Scientific Computing Group at FRM II
February 6, 2013

1 Formal treatment
1.1 Green operators and the $T$-matrix
For a particle, governed by the Schrödinger equation with Hamiltonian

$$H = H_0 + V,$$

the time-independent scattering theory formally consists of solving the eigenvalue equations:

$$H \psi_n = E_n \psi_n,$$

with $E$ the scalar energy eigenvalue of the eigenstate $\psi(E)$. If the solutions of the free (or unperturbed) Hamiltonian $H_0$ are known:

$$H_0 \psi_m = E_m \psi_m,$$

one can write the solutions of the full Hamiltonian in terms of these asymptotic states and Green operators:

$$\psi_n = \psi_m + \mathcal{G}_n^m \psi_m,$$

where the Green operators are defined as:

$$\mathcal{G}_n^m(E) = (E - H_0 + i\epsilon)^{-1},$$

$$\mathcal{G}_{n}^{m}(E) = (E - H + i\epsilon)^{-1}.$$  

In these equations, the upper index or sign refers to the state corresponding with the free state $\psi_m$, at time $t \to -\infty$ (and vice versa for the lower sign). Since the solutions of the eigenvalue equations, both for the unperturbed as for the full Hamiltonian, are dependent on the energy eigenvalue $E$, the index $n$ is assumed to include this value (and possibly other quantum numbers).

The transition amplitude between two asymptotic states is given by the $S$-matrix elements, defined as:

$$S_{nm} = \langle \psi_{nm} \rangle \langle \psi_{m} \rangle = \langle \psi_{n} \mid \psi_{m} \rangle.$$