# Contents

1 Getting Started
   1.1 Installing the application .......................... 1
   1.2 Server installation ................................ 3
   1.3 Contributing Changes .............................. 3
   1.4 License ............................................ 6
   1.5 Credits ............................................. 7

2 Tutorial
   2.1 Simple films ....................................... 9
   2.2 Tethered Polymer .................................. 16
   2.3 Composite sample ................................. 21
   2.4 Superlattice Models .............................. 23
   2.5 MLayer Models .................................... 29

3 User’s Guide
   3.1 Using Refl1D ....................................... 34
   3.2 Parameters ......................................... 34
   3.3 Data Representation .............................. 34
   3.4 Materials .......................................... 39
   3.5 Sample Representation ............................ 39
   3.6 Experiment ......................................... 42
   3.7 Fitting .............................................. 43

4 Reference
   4.1 abeles - Pure python reflectivity calculator .... 53
   4.2 cheby - Freeform - Chebyshev model ............ 53
   4.3 dist - Non-uniform samples ...................... 55
   4.4 errors - Plot sample profile uncertainty ...... 58
   4.5 experiment - Reflectivity fitness function .... 60
   4.6 fitplugin - Bumps plugin definition for reflectivity models 64
   4.7 freeform - Freeform - Parametric B-Spline ... 65
   4.8 fresnel - Pure python Fresnel reflectivity calculator 67
   4.9 garefl - Adaptor for garefl models .......... 68
   4.10 instrument - Reflectivity instrument definition 69
   4.11 magnetism - Magnetic Models .................. 75
   4.12 material - Material .............................. 77
   4.13 materialdb - Materials Database ............... 80
   4.14 model - Reflectivity Models .................. 81
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.15</td>
<td>mono - Freeform - Monotonic Spline</td>
<td>83</td>
</tr>
<tr>
<td>4.16</td>
<td>names - Public API</td>
<td>85</td>
</tr>
<tr>
<td>4.17</td>
<td>ncnrdata - NCNR Data</td>
<td>85</td>
</tr>
<tr>
<td>4.18</td>
<td>polymer - Polymer models</td>
<td>98</td>
</tr>
<tr>
<td>4.19</td>
<td>probe - Instrument probe</td>
<td>103</td>
</tr>
<tr>
<td>4.20</td>
<td>profile - Model profile</td>
<td>124</td>
</tr>
<tr>
<td>4.21</td>
<td>reflectivity - Reflectivity</td>
<td>127</td>
</tr>
<tr>
<td>4.22</td>
<td>reflmodule - Low level reflectivity calculations</td>
<td>129</td>
</tr>
<tr>
<td>4.23</td>
<td>resolution - Resolution</td>
<td>130</td>
</tr>
<tr>
<td>4.24</td>
<td>snsdata - SNS Data</td>
<td>133</td>
</tr>
<tr>
<td>4.25</td>
<td>staj - Staj File</td>
<td>137</td>
</tr>
<tr>
<td>4.26</td>
<td>stajconvert - Staj File Converter</td>
<td>143</td>
</tr>
<tr>
<td>4.27</td>
<td>stitch - Overlapping reflectivity curve stitching</td>
<td>144</td>
</tr>
<tr>
<td>4.28</td>
<td>support - Environment support</td>
<td>145</td>
</tr>
<tr>
<td>4.29</td>
<td>util - Miscellaneous functions</td>
<td>146</td>
</tr>
</tbody>
</table>

Python Module Index

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>147</td>
</tr>
</tbody>
</table>
CHAPTER 1

Getting Started

1-D reflectometry allows material scientists to understand the structure of thin films, providing composition and density information as a function of depth. With polarized neutron measurements, scientists can study the sub-surface structure of magnetic samples. The Refl1D modeling program supports a mixture of slabs, freeform and specialized layer types such as models for the density distribution of polymer brushes.

1.1 Installing the application

- Installing from source
  - Windows
  - Linux
  - OS/X

Recent versions of the Refl1D application are available for windows and mac from http://www.ncnr.nist.gov/reflpak. The installer walks through the steps of setting the program up to run on your machine and provides the sample data used in the tutorial.

Linux users will need to install from using pip:

```
pip install refl1d
```

Note that the binary versions will lag the release version until the release process is automated. Windows and Mac users may want to install using pip as well to get the version with the latest changes.

1.1.1 Installing from source

Installing the application from source requires a working python environment. See below for operating system specific instructions.

Our base scientific python environment contains the following packages. The versions listed are a snapshot of our current configuration, though both older and more recent versions are likely to work:

- python 2.7
- matplotlib 1.3.1
- numpy 1.9.0
- scipy 0.14.0
- wxPython 2.9.5.0
• setuptools 7.0
• pyparsing 1.5.6
• pip 1.4.1

Python 3.3/3.4 will work for batch processing, but wxPython is not yet supported.

Once your environment is in place, you can install directly from PyPI using pip:

```bash
pip install refl1d
```

This will install refl1d, bumps and periodictable.

You can run the program by typing:

```bash
python -m refl1d.main
```

If this fails, then follow the instructions in Contributing Changes to install from the source archive directly.

**Windows**

There are several options for setting up a python environment on windows:

• Python(X,Y),
• WinPython,
• Anaconda,
• Canopy, or
• python.org.

Python(X,Y) is easiest as of this writing, since it contains all the required packages. You can select “Full Install” for convenience, or you can select “Custom Install” and make sure the required packages are selected. In particular, wx
is not selected by default. Be sure to select py2exe and sphinx as well, since you may want to build a self contained release package. For the other environments, you may have to download and install the wxPython binary package manually.

You will need a C/C++ compiler. If you already have Microsoft Visual C installed you are done. If not, you can use the MinGW compiler that is supplied with your python environment or download your own. You can set MinGW as the default compiler by creating the file `Libdistutils\distutils.cfg` in your python directory (e.g., `C:\Python2.7`) with the following content:

```ini
[build]
compiler=mingw32
```

Once the python is prepared, you can install the periodic table and bumps package using the Windows console. To start the console, click the “Start” icon on your task bar and select “Run...”. In the Run box, type “cmd”.

**Linux**

Linux distributions will provide the base required packages. You will need to refer to your distribution documentation for details.

On debian/ubuntu, the command will be something like:

```bash
sudo apt-get install python-{matplotlib,numpy,scipy,wxgtk2.8,pyparsing,setuptools}
```

For development you also want nose and sphinx:
sudo apt-get install python-{nose,sphinx}

Latex is needed to build the pdf documentation.

**OS/X**

For OS/X you have a few options for setting up a python distribution:

- Anaconda,
- Canopy,
- MacPorts, or
- python.org.

To run the interactive interface on OS/X you will need to use:

```
pythonw -m refl1d.main --edit
```

### 1.2 Server installation

Refl-1D jobs can be submitted to a remote bumps queue for processing. You just need to install the refl1d plugin in the bumps server.

TODO: show details.

### 1.3 Contributing Changes

- **Simple patches**
- **Larger changes**
  - Building Documentation
  - Windows Installer
  - OS/X Installer

The best way to contribute to the reflectometry package is to work from a copy of the source tree in the revision control system.

The refl1d project is hosted on github at:

https://github.com/reflectometry/refl1d

You will need the git source control software for your computer. This can be downloaded from the git page, or you can use an integrated development environment (IDE) such as Eclipse and PyCharm, which may have git built in.

#### 1.3.1 Simple patches

If you want to make one or two tiny changes, it is easiest to clone the project, make the changes, document and test, then send a patch.

Clone the project as follows:
You will need bumps and periodictable to run. If you are fixing bugs in the scattering length density calculator or the fitting engine, you will want to clone the repositories as sister directories to the refl1d source tree:

git clone https://github.com/bumps/bumps.git

If you are only working with the refl1d modeling code, then you can install bumps and periodictable using pip:
pip install periodictable bumps

To run the package from the source tree use the following:
cd refl1d
python run.py

This will first build the package into the build directory then run it. Any changes you make in the source directory will automatically be used in the new version.

As you make changes to the package, you can see what you have done using git:
git status

git diff

Please update the documentation and add tests for your changes. We use doctests on all of our examples that we know our documentation is correct. More thorough tests are found in test directory. With the nosetest package, you can run the tests using:

python tests.py

Nose is available on linux form apt-get

When all the tests run, create a patch and send it to paul.kienzle@nist.gov:
git diff > patch

### 1.3.2 Larger changes

For a larger set of changes, you should fork refl1d on github, and issue pull requests for each part.

Once you have create the fork, the clone line is slightly different:
git clone https://github.com/YourGithubAccount/refl1d

After you have tested your changes, you will need to push them to your github fork:
git log

Get commit -a -m "short sentence describing what the change is for"
git push

Good commit messages are a bit of an art. Ideally you should be able to read through the commit messages and create a “what’s new” summary without looking at the actual code.

Make sure your fork is up to date before issuing a pull request. You can track updates to the original refl1d package using:
git remote add refl1d https://github.com/reflectometry/refl1d

git fetch refl1d
When making changes, you need to take care that they work on different versions of python. In particular, RHEL6, Centos6.5, Rocks and ScientificLinux all run python 2.6, most linux/windows/mac users run python 2.7, but some of the more bleeding edge distributions run 3.3/3.4. The anaconda distribution makes it convenient to maintain multiple independent environments Even better is to test against all python versions 2.6, 2.7, 3.3, 3.4:

```
pythonX.Y tests.py
pythonX.Y run.py
```

When all the tests run, issue a pull request from your github account.

### Building Documentation

Building the package documentation requires a working Sphinx installation, and latex to build the pdf. As of this writing we are using sphinx 1.2.

The command line to build the docs is as follows:

```
(cd doc && make clean html pdf)
```

You can see the result by pointing your browser to:

```
doc/_build/html/index.html
doc/_build/latex/Refl1d.pdf
```

Note that this only works with a unix-like environment for now since we are using make. On windows, you can run sphinx directly from python:

```
cd doc
python -m sphinx.__init__ -b html -d _build/doctrees . _build/html
```

ReStructured text format does not have a nice syntax for superscripts and subscripts. Units such as g·cm⁻³ are entered using macros such as |g/cm³| to hide the details. The complete list of macros is available in `doc/sphinx/rst_prolog`

In addition to macros for units, we also define cdot, angstrom and degrees unicode characters here. The corresponding latex symbols are defined in `doc/sphinx/conf.py`.

There is a bug in older sphinx versions (e.g., 1.0.7) in which latex tables cannot be created. You can fix this by changing:

```
self.body.append(self.table.colspec)
```

to:

```
self.body.append(self.table.colspec.lower())
```

in `site-packages/sphinx/writers/latex.py`.

### Windows Installer

To build a windows standalone executable with py2exe you may first need to create an empty file named `Lib\numpy\distutils\tests\__init__.py` in your python directory (usually `C:\Python2.7`). Without this file, py2exe raises an error when it is searching for the parts of the numpy package. This may be fixed on recent versions of numpy. Next, update the `__version__` tag in `refl1d/__init__.py` to mark it as your own.
Now you can build the standalone executable using:

```
python setup_py2exe
```

This creates a dist subdirectory in the source tree containing everything needed to run the application including python and all required packages.

To build the Windows installer, you will need two more downloads:

- **Inno Setup** 5.3.10 QuickStart Pack

The C++ redistributable package is needed for programs compiled with the Microsoft Visual C++ compiler, including the standard build of the Python interpreter for Windows. It is available as vcredist_x86.exe from the Microsoft Download Center. Be careful to select the version that corresponds to the one used to build the Python interpreter — different versions can have the same name. For the Python 2.6 standard build, the file is 1.7 Mb and is dated 11/29/2007. We have a copy (vcredist_x86.exe) on our website for your convenience. Save it to the \`C:\Python26` directory so the installer script can find it.

Inno Setup creates the installer executable. When installing Inno Setup, be sure to choose the ‘Install Inno Setup Preprocessor’ option.

With all the pieces in place, you can run through all steps of the build and install by changing to the top level python directory and typing:

```
python master_builder.py
```

This creates the redistributable installer refl1d-<version>-win32.exe for Windows one level up in the directory tree. In addition, source archives in zip and tar.gz format are produced as well as text files listing the contents of the installer and the archives.

**OS/X Installer**

To build a Mac OS/X standalone executable you will need the py2app package. This should already be available in your mac python environment.

Build the executable using:

```
python setup_py2app
```

This creates a .dmg file in the dist directory with the Refl1D app inside.

### 1.4 License

The DANSE/Reflectometry group relies on a large body of open source software, and so combines the work of many authors. These works are released under a variety of licenses, including BSD and LGPL, and much of the work is in the public domain. See individual files for details.

The combined work is released under the following license:

Copyright (c) 2006-2011, University of Maryland All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the
distribution. Neither the name of the University of Maryland nor the names of its contributors may be
used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS”
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PUR-
POSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBU-
TORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUB-
STITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUP-
TION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY
WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.

Additional pieces may be available under the GPL. When these pieces are used in the package, the combined work is
also subject to the GPL.

1.5 Credits

Refl1D package was developed under DANSE project and is maintained by its user community.

Please cite:


Available from http://reflectometry.org/danse

We are grateful for the existence of many fine open source packages such as Pyparsing, NumPy and Python without
which this package would be much more difficult to write.
This tutorial will describe walk through the steps of setting up a model with Python scripting. Scripting allows the user to create complex models with many constraints relatively easily.

2.1 Simple films

These tutorials describe the process of defining reflectometry depth profiles using scripts. Scripts are defined using Python. Python is easy enough that you should be able to follow the tutorial and use one of our examples as a starting point for your own models. A complete introduction to programming and Python is beyond the scope of this document, and the reader is referred to the many fine tutorials that exist on the web.

2.1.1 Defining a film

We start with a basic example, a nickel film on silicon:
This model shows three layers (silicon, nickel, and air) as seen in the solid green line (the step profile). In addition we have a dashed green line (the smoothed profile) which corresponds the effective reflectivity profile, with the \( \exp(-2k_nk_{n+1}\sigma^2) \) interface factored in.

This model is defined in nifilm.py.

You can preview the model on the command line:

```
$ refl1d nifilm.py --preview
```

Let’s examine the code down on a line by line basis to understand what is going on.

The first step in any model is to load the names of the functions and data that we are going to use. These are defined in a module named refl1d.names, and we import them all as follows:

```python
from refl1d.names import *
```

This statement imports functions like SLD and Material for defining materials, Parameter, Slab and Stack for defining materials, NeutronProbe and XrayProbe for defining data, and Experiment and FitProblem to tie everything together.

Note that ‘import *’ is bad style for anything but simple scripts. As programs get larger, it is much less confusing to list the specific functions that you need from a module rather than importing everything.

Next we define the materials that we are going to use in our sample. silicon and air are common, so we don’t need to define them. We just need to define nickel, which we do as follows:

```python
nickel = Material('Ni')
```
This defines a chemical formula, Ni, for which the program knows the density in advance since it has densities for all elements. By using chemical composition, we can compute scattering length densities for both X-ray and neutron beams from the same sample description. Alternatively, we could take a more traditional approach and define nickel as a specific SLD for our beam.

```
#nickel = SLD(rho=9.4)
```

The ‘#’ character on the above line means that line is a comment, and it won’t be evaluated.

With our materials defined (silicon, nickel and air), we can combine them into a sample. The substrate will be silicon with a 5 Å 1-σ Si:Ni interface. The nickel layer is 100 Å thick with a 5 Å Ni:Air interface. Air is on the surface.

```
sample = silicon(0,5) | nickel(100,5) | air
```

Our sample definition is complete, so now we need to specify the range of values we are going to view. We will use the `numpy` library, which extends python with vector and matrix operations. The `linspace` function below returns values from 0 to 5 in 100 steps for incident angles from 0° to 5°.

```
T = numpy.linspace(0, 5, 100)
```

From the range of reflection angles, we can create a neutron probe. The probe defines the wavelengths and angles which are used for the measurement as well as their uncertainties. From this the resolution of each point can be calculated. We use constants for angular divergence $dT=0.01°$, wavelength $L=4.75$ Å and wavelength dispersion $dL=0.0475$ in this example, but each angle and wavelength is independent.

```
probe = NeutronProbe(T=T, dT=0.01, L=4.75, dL=0.0475)
```

Combine the neutron probe with the sample stack to define an experiment. Using chemical formula and mass density, the same sample can be simulated for both neutron and x-ray experiments.

```
M = Experiment(probe=probe, sample=sample)
```

Generate a random data set with 5% noise. While not necessary to display a reflectivity curve, it is useful in showing how the data set should look.

```
M.simulate_data(5)
```

Combine a set of experiments into a fitting problem. The problem is used by refl1d for all operations on the model.

```
problem = FitProblem(M)
```

### 2.1.2 Choosing an instrument

Let’s modify the simulation to show how a 100 Å nickel film might look if measured on the SNS Liquids reflectometer:

```
2.1. Simple films
```

11
This model is defined in nifilm-tof.py

The sample definition is the same:

```python
from refl1d.names import *

nickel = Material('Ni')
sample = silicon(0,5) | nickel(100,5) | air
```

Instead of using a generic probe, we are using an instrument definition to control the simulation.

```python
instrument = SNS.Liquids()
M = instrument.simulate(sample,
    T=[0.3,0.7,1.5,3],
    slits=[0.06, 0.14, 0.3, 0.6],
    uncertainty = 5,
)
```

The `instrument` line tells us to use the geometry of the SNS Liquids reflectometer, which includes information like the distance between the sample and the slits and the wavelength range. We then simulate measurements of the sample for several different angles $T$ (degrees), each with its own slit opening $slits$ (mm). The simulated measurement duration is such that the median relative error on the measurement $\Delta R/R$ will match $uncertainty$ ($\%$). Because the intensity $I(\lambda)$ varies so much for a time-of-flight measurement, the central points will be measured with much better precision, and the end points will be measured with lower precision. See `Pulsed.simulate` for details on all simulation parameters.

Finally, we bundle the simulated measurement as a fit problem which is used by the rest of the program.
2.1.3 Attaching data

Simulating data is great for seeing how models might look when measured by a reflectometer, but mostly we are going to use the program to fit measured data. We saved the simulated data from above into files named nifilm-tof-1.dat, nifilm-tof-2.dat, nifilm-tof-3.dat and nifilm-tof-4.dat. We can load these datasets into a new model using nifilm-data.py.

The sample and instrument definition is the same as before:

```python
from refl1d.names import *

nickel = Material('Ni')
sample = silicon(0,5) | nickel(100,5) | air

instrument = SNS.Liquids()
```

In this case we are loading multiple data sets into the same `ProbeSet` object. If your reduction program stitches together the data for you, then you can simply use `probe=instrument.load('file')`.

```python
files = ['nifilm-tof-%d.dat' % d for d in (1,2,3,4)]
probe = ProbeSet(instrument.load(f) for f in files)
```

The data and sample are combined into an `Experiment`, which again is bundled as a `FitProblem` for the fitting program.

```python
M = Experiment(probe=probe, sample=sample)
problem = FitProblem(M)
```

The plot remains the same:
2.1.4 Performing a fit

Now that we know how to define a sample and load data, we can learn how to perform a fit on the data. This is shown in nifilm-fit.py:

We use the usual sample definition, except we set the thickness of the nickel layer to 125 Å so that the model does not match the data:

```python
from refl1d.names import *

nickel = Material('Ni')
sample = silicon(0,10) | nickel(125,10) | air
```

We are going to try to recover the original thickness by letting the thickness value range by 125 ± 50 Å. Since nickel is layer 1 in the sample (counting starts at 0 in Python), we can access the layer parameters using sample[1]. The parameter we are accessing is the thickness parameter, and we are setting it’s fit range to ±50 Å.

```python
sample[1].thickness.pm(50)
```

We are also going to let the interfacial roughness between the layers vary. The interface between two layers is defined by the width of the interface on top of the layer below. Here we are restricting the silicon:nickel interface to the interval [3, 12] and the nickel:air interface to the range [0, 20]:

```python
sample[0].interface.range(3,12)
sample[1].interface.range(0,20)
```
The data is loaded as before.

```
instrument = SNS.Liquids()
files = ['nifilm-tof-%d.dat' % d for d in (1,2,3,4)]
probe = ProbeSet(instrument.load(f) for f in files)
M = Experiment(probe=probe, sample=sample)
problem = FitProblem(M)
```

As you can see the new nickel thickness changes the theory curve significantly:

![Plot showing the effect of thickness on the theory curve](image-url)

We can now load and run the fit:

```
$ refl1d nifilm-fit.py --fit=newton --steps=100 --store=T1
```

The `--fit=newton` option says to use the quasi-newton optimizer for not more than 100 steps. The `--store=T1` option says to store the initial model, the fit results and any monitoring information in the directory T1.

Here is the resulting fit:

All is well: Normalized $\chi^2_N$ is close to 1 and the line goes nicely through the data.
2.1.5 Back reflectivity

For samples measured with the incident beam through the substrate rather than reflecting off the surface, we don’t need to modify our sample, we just need to tell the experiment that we are measuring back reflectivity.

We set up the example as before.

```python
from refl1d.names import *
nickel = Material('Ni')
sample = silicon(0,25) | nickel(100,5) | air
T = numpy.linspace(0, 5, 100)
```

Because we are measuring back reflectivity, we create a probe which has `back_reflectivity = True`.

```python
probe = NeutronProbe(T=T, dT=0.01, L=4.75, dL=0.0475, back_reflectivity=True)
```

The remainder of the model definition is unchanged.

```python
M = Experiment(probe=probe, sample=sample)
M.simulate_data(5)
problem = FitProblem(M)
```

2.2 Tethered Polymer

Soft matter systems have more complex interfaces than slab layers with gaussian roughness.

We will now model a data set for tethered deuterated polystyrene chains. The chains start out at approximately 10 nm thick in dry conditions, and swell to 14-18 nm thickness in toluene. Two measurements were made:

- 10ndt001.refl in deuterated toluene
- 10nht001.refl in hydrogenated toluene

The chains are bound to the substrate by an initiator layer between the substrate and brush chains. So the model needs a silicon layer, silicon oxide layer, an initiator layer which is mostly hydrocarbon and scattering length density should be between 0 and 1.5 depending on how much solvent is in the layer. Then you have the swollen brush chains and at the end bulk solvent. For these swelling measurements, the beam penetrate the system from the silicon side and the bottom layer is deuterated or hydrogenated toluene.

2.2.1 Defining the film

We first need to define the materials

```python
from refl1d.names import *
from copy import copy

# === Materials ===
SiOx = SLD(name="SiOx",rho=3.47)
D_toluene = SLD(name="D-toluene",rho=5.66)
D_initiator = SLD(name="D-initiator",rho=1.5)
D_polystyrene = SLD(name="D-PS",rho=6.2)
H_toluene = SLD(name="H-toluene",rho=0.94)
H_initiator = SLD(name="H-initiator",rho=0)
```

In this case we are using the neutron scattering length density as is standard practice in reflectivity experiments rather than the chemical formula and mass density. The `SLD` class allows us to name the material and define the real and
imaginary components of scattering length density \( \rho \). Note that we are using the imaginary \( \rho_i \) rather than the absorption coefficient \( \mu = 2\lambda \rho_i \) since it removes the dependence on wavelength from the calculation of the reflectivity.

For the tethered polymer we don’t use a simple slab model, but instead define a \texttt{PolymerBrush} layer, which understands that the system is composed of polymer plus solvent, and that the polymer chains tail off like:

\[
V(z) = \begin{cases} 
V_o & \text{if } z \leq z_o \\
V_o(1 - ((z - z_o)/(L))^\rho) & \text{if } z_o < z < z_o + L \\
0 & \text{if } z \geq z_o + L 
\end{cases}
\]

This volume profile combines with the scattering length density of the polymer and the solvent to form an SLD profile:

\[
\rho(z) = \rho_p V(z) + \rho_s (1 - V(z))
\]

The tethered polymer layer definition looks like

```python
# === Sample ===
# Deuterated sample
D_brush = PolymerBrush(polymer=D_polystyrene, solvent=D_toluene,
                        base_vf=70, base=120, length=80, power=2,
                        sigma=10)
```

This layer can be combined with the remaining layers to form the deuterated measurement sample

```python
D = (silicon(0,5) | SiOx(100,5) | D_initiator(100,20) | D_brush(400,0)
    | D_toluene)
```

The undeuterated sample is similar to the deuterated sample. We start by copying the polymer brush layer so that parameters such as \texttt{length}, \texttt{power}, etc. will be shared between the two systems, but we replace the deuterated toluene solvent with undeuterated toluene. We then use this \texttt{H_brush} to define a new stack with undeuterated toluene

```python
# Undeuterated sample is a copy of the deuterated sample
H_brush = copy(D_brush)  # Share tethered polymer parameters...
H_brush.solvent = H_toluene  # ... but use different solvent
H = silicon | SiOx | H_initiator | H_brush | H_toluene
```

We want to share thickness and interface between the two systems as well, so we write a loop to go through the layers of \( D \) and copy the thickness and interface parameters to \( H \)

```python
for i, _ in enumerate(D):
    H[i].thickness = D[i].thickness
    H[i].interface = D[i].interface
```

What is happening internally is that for each layer in the stack we are copying the parameter for the thickness from the deuterated sample slab to the thickness slot in the undeuterated sample slab. Similarly for interface. When the refinement engine sets a new value for a thickness parameter and asks the two models to evaluate \( \chi^2 \), both models will see the same thickness parameter value.
2.2.2 Setting fit ranges

With both samples defined, we next specify the ranges on the fitted parameters

```python
# === Fit parameters ===
for i in 0, 1, 2:
    D[i].interface.range(0, 100)
D[1].thickness.range(0, 200)
D[2].thickness.range(0, 200)
D_polystyrene.rho.range(6.2, 6.5)
SiOx.rho.range(2.07, 4.16) # Si to SiO2
D_toluene.rho.pmp(5)
D_initiator.rho.range(0, 1.5)
D_brush.base_vf.range(50, 80)
D_brush.base.range(0, 200)
D_brush.length.range(0, 500)
D_brush.power.range(0, 5)
D_brush.sigma.range(0, 20)

# Undeuterated system adds two extra parameters
H_toluene.rho.pmp(5)
H_initiator.rho.range(-0.5, 0.5)
```

Notice that in some cases we are using layer number to reference the parameter, such as `D[1].thickness` whereas in other cases we are using variables directly, such as `D_toluene.rho`. Determining which to use requires an understanding of the underlying stack model. In this case, the thickness is associated with the SiOx slab thickness, but we never formed a variable to contain `Slab(material=SiOx)`, so we have to reference it via the stack. We did however create a variable to contain `Material(name="D_toluene")` so we can access its parameters directly. Also, notice that we only need to set one of `D[1].thickness` and `H[1].thickness` since they are the same underlying parameter.

2.2.3 Attaching data

Next we associate the reflectivity curves with the samples:

```python
# === Data files ===
instrument = NCNR.NG7(Qlo=0.005, slits_at_Qlo=0.075)
D_probe = instrument.load('10ndt001.refl', back_reflectivity=True)
H_probe = instrument.load('10nht001.refl', back_reflectivity=True)
D_probe.theta_offset.range(-0.1, 0.1)
```

We set `back_reflectivity=True` because we are coming in through the substrate. The reflectometry calculator will automatically reverse the stack and adjust the effective incident angle to account for the refraction when the beam enters the side of the substrate. Ideally you will have measured the incident beam intensity through the substrate as well so that substrate absorption effects are corrected for in your data reduction steps, but if not, you can set an estimate for `back_absorption` when you load the file. Like `intensity` you can set a range on the value and adjust it during refinement.

Finally, we define the fitting problem from the probes and samples. The `dz` parameter controls the size of the profiles when generating the tethered polymer interface. The `dA` parameter allows these steps to be joined together into larger slabs, with each slab having $(\rho_{\text{max}} - \rho_{\text{min}})w < \Delta A$.

```python
# === Problem definition ===
D_model = Experiment(sample=D, probe=D_probe, dz=0.5, dA=1)
H_model = Experiment(sample=H, probe=H_probe, dz=0.5, dA=1)
models = H_model, D_model
```
This is a multifit problem where both models contribute to the goodness of fit measure $\chi^2$. Since no weight vector was defined the fits have equal weight.

```python
problem = FitProblem(models)
problem.name = "tethered"
```

The polymer brush model is a smooth profile function, which is evaluated by slicing it into thin slabs, then joining together similar slabs to improve evaluation time. The $dz=0.5$ parameter tells us that we should slice the brush into 0.5 Å steps. The $dA=1$ parameter says we should join together thin slabs while the scattering density uncertainty in the joined slabs $\Delta A < 1$, where $\Delta A = (\max \rho - \min \rho)(\max z - \min z)$. Similarly for the absorption cross section $\rho_i$ and the effective magnetic cross section $\rho_M \cos(\theta_M)$. If $dA=None$ (the default) then no profile contraction occurs.

The resulting model looks like:

![Graph](image-url)
This complete model script is defined in `tethered.py`:

```python
from refl1d.names import *
from copy import copy

# === Materials ===
SiOx = SLD(name="SiOx", rho=3.47)
D_toluene = SLD(name="D-toluene", rho=5.66)
D_initiator = SLD(name="D-initiator", rho=1.5)
D_polystyrene = SLD(name="D-PS", rho=6.2)
H_toluene = SLD(name="H-toluene", rho=0.94)
H_initiator = SLD(name="H-initiator", rho=0)

# === Sample ===
# Deuterated sample
D_brush = PolymerBrush(polymer=D_polystyrene, solvent=D_toluene,
                        base_vf=70, base=120, length=80, power=2,
                        sigma=10)
D = (silicon(0,5) | SiOx(100,5) | D_initiator(100,20) | D_brush(400,0) |
     D_toluene)

# Undeuterated sample is a copy of the deuterated sample
H_brush = copy(D_brush)    # Share tethered polymer parameters...
H_brush.solvent = H_toluene  # ... but use different solvent
H = silicon | SiOx | H_initiator | H_brush | H_toluene
```
for i, _ in enumerate(D):
    H[i].thickness = D[i].thickness
    H[i].interface = D[i].interface

# === Fit parameters ===
for i in 0, 1, 2:
    D[i].interface.range(0, 100)
    D[1].thickness.range(0, 200)
    D[2].thickness.range(0, 200)
    D_polystyrene.rho.range(6.2, 6.5)
    SiOx.rho.range(2.07, 4.16)  # Si to SiO2
    D_toluene.rho.pmp(5)
    D_initiator.rho.range(0, 1.5)
    D_brush.base_vf.range(50, 80)
    D_brush.base.range(0, 200)
    D_brush.length.range(0, 500)
    D_brush.power.range(0, 5)
    D_brush.sigma.range(0, 20)

# Undeuterated system adds two extra parameters
H_toluene.rho.pmp(5)
H_initiator.rho.range(-0.5, 0.5)

# === Data files ===
instrument = NCNR.NG7(Qlo=0.005, slits_at_Qlo=0.075)
D_probe = instrument.load('10ndt001.refl', back_reflectivity=True)
H_probe = instrument.load('10nht001.refl', back_reflectivity=True)

D_probe.theta_offset.range(-0.1, 0.1)

# === Problem definition ===
D_model = Experiment(sample=D, probe=D_probe, dz=0.5, dA=1)
H_model = Experiment(sample=H, probe=H_probe, dz=0.5, dA=1)
models = H_model, D_model
problem = FitProblem(models)
problem.name = "tethered"

The model can be fit using the parallel tempering optimizer:

$ refl1d tethered.py --fit=pt --store=T1

2.3 Composite sample

There are conditions wherein the sample you measure is not ideal. For example, a polymer brush may have enough density in some domains that the brushes are standing upright, but in other domains the brushes lie flat.

2.3.1 Channel measurement

In this example we will look at a nickel grating on a silicon substrate using specular reflectivity. When the spacing within the grating is sufficiently large, this can be modeled to first order as the incoherent sum of the reflectivity on the plateau and the reflectivity on the valley floor. By adjusting the weight of two reflectivities, we should be able to determine the ratio of plateau width to valley width.
Since silicon and air are defined, the only material we need to define is nickel.

```python
from refl1d.names import *
nickel = Material('Ni')
```

We need two separate models, one with 1000 Å nickel and one without.

```python
plateau = silicon(0,5) | nickel(1000,200) | air
valley = silicon(0,5) | air
```

We need only one probe for simulation. The reflectivity measured at the detector will be a mixture of those neutrons which reflect off the plateau and those that reflect off the valley.

```python
T = numpy.linspace(0, 2, 200)
probe = NeutronProbe(T=T, dT=0.01, L=4.75, dL=0.0475)
```

We are going to start with a 1:1 ratio of plateau to valley and create a simulated data set.

```python
M = MixedExperiment(samples=[plateau,valley], probe=probe, ratio=[1,1])
M.simulate_data(5)
```

We will assume the silicon interface is the same for the valley as the plateau, which depending on the how the sample is constructed, may or may not be realistic.

```python
valley[0].interface = plateau[0].interface
```

We will want to fit the thicknesses and interfaces as usual.
The ratio between the valley and the plateau can also be fit, either by fixing size of the plateau and fitting the size of the valley or fixing the size of the valley and fitting the size of the plateau. We will hold the plateau fixed.

```
M.ratio[1].range(0,5)
```

Note that we could include a second order effect by including a hillside term with the same height as the plateau but using a 50:50 mixture of air and nickel. In this case we would have three entries in the ratio.

We wrap this as a fit problem as usual.

```
problem = FitProblem(M)
```

This complete model script is defined in `mixed.py`:

```python
from refl1d.names import *

nickel = Material('Ni')

plateau = silicon(0,5) | nickel(1000,200) | air
valley = silicon(0,5) | air

T = numpy.linspace(0, 2, 200)
probe = NeutronProbe(T=T, dT=0.01, L=4.75, dL=0.0475)

M = MixedExperiment(samples=[plateau,valley], probe=probe, ratio=[1,1])
M.simulate_data(5)

valley[0].interface = plateau[0].interface

plateau[0].interface.range(0,200)
plateau[1].interface.range(0,200)
plateau[1].thickness.range(200,1800)

M.ratio[1].range(0,5)

problem = FitProblem(M)
```

We can test how well the fitter can recover the original model by running refl1d with `--random`:

```
$ refl1d mixed.py --random --store=T1
```

## 2.4 Superlattice Models

Any structure can be turned into a superlattice using a `refl1d.model.Repeat`.

Simply form a stack as usual, then use that stack within another stack, with a repeat modifier.

### 2.4.1 Hard material structures

Here is an example of a multilayer system in the literature:

In this paper, the authors are interested in the interdiffusion properties of Ni into Ti through x-ray and neutron reflectivity measurements. The question of alloying at metal-metal interfaces at elevated temperatures is critically important for device fabrication and reliability.

The model is defined in NiTi.py.

First define the materials we will use

```python
from refl1d.names import *
nickel = Material('Ni')
titanium = Material('Ti')
```

Next we will compose nickel and titanium into a bilayer and use that bilayer to define a stack with 10 repeats.

```python
# Superlattice description
bilayer = nickel(50,5) | titanium(50,5)
sample = silicon(0,5) | bilayer*10 | air
```

We allow the thickness to vary by +/- 100%.

```python
# Fitting parameters
bilayer[0].thickness.pmp(100)
bilayer[1].thickness.pmp(100)
```

The interfaces vary between 0 and 30 Å. The interface between repeats is defined by the interface at the top of the repeating stack, which in this case is the Ti interface. The interface between the superlattice and the next layer is an independent parameter, whose value defaults to the same initial value as the interface between the repeats.
If we wanted to have the interface for Ti between repeats identical to the interface between Ti and air, we could have tied the parameters together, but we won’t in this example:

```python
# sample[1].interface = bilayer[1].interface
```

If instead we wanted to keep the roughness independent, but start with a different initial value, we could simply set the interface parameter value. In this case, we are setting it to 10 Å

```python
# sample[1].interface.value = 10
```

We can also fit the number of repeats. This is not realistic in this example (the sample grower surely knows the number of layers in a sample like this), so we do so only to demonstrate how it works.

```python
sample[1].repeat.range(5,15)
```

Before we can view the reflectivity, we must define the Q range over which we want to simulate, and combine this probe with the sample.

```python
T = numpy.linspace(0, 5, 100)
probe = XrayProbe(T=T, dT=0.01, L=4.75, dL=0.0475)
M = Experiment(probe=probe, sample=sample)
M.simulate_data(5)
problem = FitProblem(M)
```

### 2.4.2 Soft material structures

Inter-diffusion properties of multilayer systems are of great interest in both hard and soft materials. Jomaa, et. al have shown that reflectometry can be used to elucidate the kinetics of a diffusion process in polyelectrolytes multilayers. Although the purpose of this paper was not to fit the presented system, it offers a good model for an experimentally relevant system for which information from neutron reflectometry can be obtained. In this model system we will show that we can create a model for this type of system and determine the relevant parameters through our optimisation scheme. This particular example uses deuterated reference layers to determine the kinetics of the overall system.

Reference: Jomaa, H., Schlenoff, Macromolecules, 38 (2005), 8473-8480 http://dx.doi.org/10.1021/ma050072g

We will model the system described in figure 2 of the reference as `PEMU.py`. 

---

2.4. Superlattice Models
Bring in all of the functions from refl1d.names so that we can use them in the remainder of the script.

```python
from refl1d.names import *
```

The polymer system is deposited on a gold film with chromium as an adhesion layer. Because these are standard films which are very well-known in this experiment we can use the built-in materials library to create these layers.

```python
# == Sample definition ==
chrome = Material('Cr')
gold = Material('Au')
```

The polymer system consists of two polymers, deuterated and non-deuterated PDADMA/PSS. Since the neutron scattering cross section for deuterium is considerably different from that for hydrogen while having nearly identical chemical properties, we can use the deuterium as a tag to see to what extent the deuterated polymer layer interdiffuses with an underated polymer layer.

We model the materials using scattering length density (SLD) rather than using the chemical formula and mass density. This allows us to fit the SLD directly rather than making assumptions about the specific chemical composition of the mixture.

```python
PDADMA_dPSS = SLD(name = 'PDADMA dPSS', rho = 2.77)
PDADMA_PSS = SLD(name = 'PDADMA PSS', rho = 1.15)
```

The polymer materials are stacked into a bilayer, with thickness estimates based on ellipsometery measurements (as stated in the paper).
The bilayer is repeated 5 times and stacked on the chromium/gold substrate. In this system we expect the kinetics of the surface diffusion to differ from that of the bulk layer structure. Because we want the top bilayer to optimise independently of the other bilayers, the fifth layer was not included in the stack. If the diffusion properties of each layer were expected to vary widely from one another, the repeat notation could not have been used at all.

Now that the model sample is built, we can start adding ranges to the fit parameters. We assume that the chromium and gold layers are well known through other methods and will not fit it; however, additional optimisation could certainly be included here.

As stated earlier, we will be fitting the SLD of the polymers directly. The range for each will vary from that for pure deuterated to the pure undeuterated SLD.

We are primarily interested in the interfacial roughness so we will fit those as well. First we define the interfaces within the repeated stack. Note that the interface for bilayer[1] is the interface between the current bilayer and the next bilayer. Here we use sample[3] as the repeated bilayer, which is the 0-origin index of the bilayer in the stack.

The interface between the stack and the next layer is controlled from the repeated bilayer.

Because the top bilayer has different dynamics, we optimize the interfaces independently. Although we want the optimiser to treat these parameters independently because surface diffusion is expected to occur faster, the overall nature of the diffusion is expected to be the same and so we use the same limits.

Finally we need to associate the sample with a measurement. We do not have the measurements from the paper available, so instead we will simulate a measurement but setting up a neutron probe whose incident angles range from 0 to 5 degrees in 100 steps. The simulated measurement is returned together with the model as a fit problem.

# 2.4.3 Freeform structures

The following is a freeform superlattice floating in a solvent and anchored with a tether molecule. The tether is anchored via a thiol group to a multilayer of Si/Cr/Au. The sulphur in the thiol attaches well to gold, but not silicon. Gold will stick to chrome which sticks to silicon.
Here is the plot using a random tether, membrane and tail group:

The model is defined by `freeform.py`.

The materials are straightforward:

```python
from refl1d.names import *

chrome = Material('Cr')
gold = Material('Au')
solvent = Material('H2O', density=1)
```

The sample description is more complicated. When we define a freeform layer we need to anchor the ends of the freeform layer to a known material. Usually, this is just the material that makes up the preceding and following layer. In case we have freeform layers connected to each other, though, we need an anchor material that controls the SLD at the connection point. For this purpose we introduce the dummy material wrap:

```python
wrap = SLD(name="wrap", rho=0)
```

Each section of the freeform layer has a different number of control points. The value should be large enough to give the profile enough flexibility to match the data, but not so large that it over fits the data. Roughly the number of control points is the number of peaks and valleys allowed. We want a relatively smooth tether and tail, so we keep \( n_1 \) and \( n_3 \) small, but make \( n_2 \) large enough to define an interesting repeat structure.

```python
n1, n2, n3 = 3, 9, 3
```

Free layers have a thickness, horizontal control points \( z \) varying in \([0, 1]\), real and complex SLD \( \rho \) and \( \rho_i \), and the material above and below.
Refl1D: Neutron and X-Ray Reflectivity Analysis, Release 0.7.8

```python
tether = FreeLayer(below=gold, above=wrap, thickness=10,
                   z=numpy.linspace(0,1,n1+2)[1:-1],
                   rho=numpy.random.rand(n1), name="tether")
bilayer = FreeLayer(below=wrap, above=wrap, thickness=80,
                   z=numpy.linspace(0,1,n2+2)[1:-1],
                   rho=5*numpy.random.rand(n2)-1, name="bilayer")
tail = FreeLayer(below=wrap, above=solvent, thickness=10,
                 z=numpy.linspace(0,1,n3+2)[1:-1],
                 rho=numpy.random.rand(n3), name="tail")
```

With the predefined free layers, we can quickly define a stack, with the bilayer repeat structure. Note that we are setting the thickness for the free layers when we define the layers, so there is no need to set it when composing the layers into a sample.

```python
sample = (silicon(0,5) | chrome(20,2) | gold(50,5)
         | tether | bilayer*10 | tail | solvent)
```

Finally, simulate the resulting model.

```python
T = numpy.linspace(0, 5, 100)
probe = NeutronProbe(T=T, dT=0.01, L=4.75, dL=0.0475,
                     back_reflectivity=True)
M = Experiment(probe=probe, sample=sample, dA=5)
M.simulate_data(5)
problem = FitProblem(M)
```

### 2.5 MLayer Models

This package can load models from other reflectometry fitting software. In this example we load an mlayer .staj file and fit the parameters within it.

The staj file can be used directly from the graphical interactor or it can be previewed from the command line:

```bash
$ refl1d De2_VATR.staj --preview
```

This shows the model plot:
and the available model parameters:

```plaintext
.probe
  .back_absorption = Parameter(1, name='back_absorption')
  .background = Parameter(1e-10, name='background')
  .intensity = Parameter(1, name='intensity')
  .theta_offset = Parameter(0, name='theta_offset')

.sample
  .layers
    [0]
      .interface = Parameter(4.24661e-11, name='B3 interface')
      .material
        .irho = Parameter(3.00904e-05, name='B3 irho')
        .rho = Parameter(5.69228, name='B3 rho')
        .thickness = Parameter(90, name='B3 thickness')
    [1]
      .interface = Parameter(4.24661e-11, name='B2 interface')
      .material
        .irho = Parameter(1.39368e-05, name='B2 irho')
        .rho = Parameter(5.86948, name='B2 rho')
        .thickness = Parameter(64.0154, name='B2 thickness')
    [2]
      .interface = Parameter(83.7958, name='B1 interface')
      .material
        .irho = Parameter(6.93684e-05, name='B1 irho')
        .rho = Parameter(0.340309, name='B1 rho')
```

30 Chapter 2. Tutorial
Note that the parameters are reversed from the order in mlayer, so layer 0 is the substrate rather than the incident medium. The graphical interactor, refl1d_gui, allows you to adjust parameters and fit ranges before starting the fit, but you can also do so from a script, as shown in De2_VATR.py:

```
from refl1d.names import *
from refl1d.stajconvert import load_mlayer

# Load neutron model and data from staj file
# Layer names are ordered from substrate to surface, and defaults to
# the names in the original staj file.
# Model name defaults to the data file name
layers=['sappire','MgO','MgHx1','MgHx2','Pd','air']
M = load_mlayer("De2_VATR.staj",layers=layers,name="n6hd2")

# Set thickness/roughness fitting parameters to +/- 20 %
# Set SLD to +/- 5% for all but the incident medium and the substrate.
for L in M.sample[1:-1]:
    L.thickness.pmp(20)
    L.interface.pmp(20)
    L.material.rho.pmp(5)

# Let the substrate SLD vary by 2%
M.sample[0].material.rho.pmp(2)
M.sample[0].interface.range(0,20)
M.sample[1].interface.range(0,20)

problem = FitProblem(M)
problem.name = "Desorption 2"
```

Staj file constraints are ignored, but you can get similar functionality by setting parameters to equal expressions of other parameters. You can even constrain one staj file to share parameters with another by setting, for example:

```
M1 = load_mlayer("De1_VATR.staj")
M2 = load_mlayer("De2_VATR.staj")
M1.sample[3].thickness = M2.sample[3].thickness
problem = MultiFitProblem([M1,M2])
```
Refl1D is a complex piece of software hiding some simple mathematics. The reflectivity of a sample is a simple function of its optical transform matrix $M$. By slicing the sample in uniform layers, each of which has a transfer matrix $M_i$, we can estimate the transfer matrix for a depth-varying sample using $M = \prod M_i$. We can adjust the properties of the individual layers until the measured reflectivity best matches the calculated reflectivity.

The complexity comes from multiple sources:

- Determining depth structure from reflectivity is an inverse problem requiring a search through a landscape with multiple minima, whose global minimum is small and often in an unpromising region.
- The solution is not unique: multiple minima may be equally valid solutions to the inversion problem.
- The measurement is sensitive to nuisance parameters such as sample alignment. That means the analysis program must include data reduction steps, making data handling complicated.
- The models are complex. Since the ideal profile is not unique and is difficult to locate, we often constrain our search to feasible physical models to limit the search space, and to account for information from other sources.
- The reflectivity is dependent on the type of radiation used to probe the sample and even its energy.

Using Refl1D

Model scripts associate a sample description with data and fitting options to define the system you wish to refine.

Parameters

The adjustable values in each component of the system are defined by Parameter objects. When you set the range on a parameter, the system will be able to automatically adjust the value in order to find the best match between theory and data.

Data Representation

Data is loaded from instrument specific file formats into a generic Probe. The probe object manages the data view and by extension, the view of the theory. The probe object also knows the measurement resolution, and controls the set of theory points that must be evaluated in order to computed the expected value at each point.

Materials

The strength of the interaction can be represented either in terms of their scattering length density using SLD, or by their chemical formula using Material, with scattering length density computed from the information in the probe. Mixture can be used to make a composite material whose parts vary be mass or by volume.

Sample Representation
Materials are composed into samples, usually as a Stack of Slabs layers, but more specific profiles such as PolymerBrush are available. Freeform sections of the profile can be described using FreeLayer, allowing arbitrary scattering length density profiles within the layer, or FreeInterface allowing arbitrary transitions from one SLD to another. New layer types can be defined by subclassing Layer.

Experiment

Sample descriptions and data sets are combined into an Experiment object, allowing the program to compute the expected reflectivity from the sample and the probability that reflectivity measured could have come from that sample. For complex cases, where the sample varies on a length scale larger than the coherence length of the probe, you may need to model your measurement with a CompositeExperiment.

Fitting

One or more experiments can be combined into a FitProblem. This is then given to one of the many fitters, such as PTFit, which adjust the varying parameters, trying to find the best fit. PTFit can also be used for Bayesian analysis in order to estimate the confidence in which the parameter values are known.

3.1 Using Refl1D

The Refl1D library is organized into modules. Specific functions and classes can be imported from a module, such as:

```python
>>> from refl1d.model import Slab
```

The most common imports have been gathered together in refl1d.names. This allows you to use names like Slab directly:

```python
>>> from refl1d.names import *
>>> s = Slab(silicon, thickness=100, interface=10)
```

This pattern of importing all names from a file, while convenient for simple scripts, makes the code more difficult to understand later, and can lead to unexpected results when the same name is used in multiple modules. A safer, though more verbose pattern is to use:

```python
>>> import refl1d.names as ref
>>> s = ref.Slab(ref.silicon, thickness=100, interface=10)
```

This documents to the reader unfamiliar with your code (such as you when looking at your model files two years from now) exactly where the name comes from.

3.2 Parameters

3.3 Data Representation
Data is represented using *Probe* objects. The probe defines the Q values and the resolution of the individual measurements, returning the scattering factors associated with the different materials in the sample. If the measurement has already been performed, the probe stores the measured reflectivity and its estimated uncertainty.

Probe objects are independent of the underlying instrument. When data is loaded, it is converted to angle \((\theta, \Delta\theta)\), wavelength \((\lambda, \Delta\lambda)\) and reflectivity \((R, \Delta R)\), with *NeutronProbe* used for neutron radiation and *XrayProbe* used for X-ray radiation. Additional properties, knowing the angle is necessary to correct for errors in sample alignment.

### 3.3.1 Simulated probes

### 3.3.2 Loading data

For time-of-flight measurements, each angle should be represented as a different probe. This eliminates the ‘stitching’ problem, where \(Q = 4\pi \sin(\theta_1)/\lambda_1 = 4\pi \sin(\theta_2)/\lambda_2\) for some \((\theta_1, \lambda_1)\) and \((\theta_2, \lambda_2)\). With stitching, it is impossible to account for effects such as alignment offset since two nominally identical Q values will in fact be different. No information is lost treating the two data sets separately — each points will contribute to the overall cost function in accordance with its statistical weight.

### 3.3.3 Viewing data

The probe object controls the plotting of theory and data curves. This is reasonable since it is only the probe which knows details such as the original points and the points used in the calculation.

### 3.3.4 Instrument Resolution

With the instrument in a given configuration \((\theta_i = \theta_f, \lambda)\), each neutron that is received is assigned to a particular Q based on the configuration. However, these values are only nominal. For example, a monochromator lets in a range of wavelengths, and slits permit a range of angles. In effect, the reflectivity measured at the configuration corresponds to a range of Q.

For monochromatic instruments, the wavelength resolution is fixed and the angular resolution varies. For polychromatic instruments, the wavelength resolution varies and the angular resolution is fixed. Resolution functions are defined in *refl1d.resolution*.

The angular resolution is determined by the geometry (slit positions, openings and sample profile) with perhaps an additional contribution from sample warp. For monochromatic instruments, measurements are taken with fixed slits at low angles until the beam falls completely onto the sample. Then as the angle increases, slits are opened to preserve full illumination. At some point the slit openings exceed the beam width, and thus they are left fixed for all angles above this threshold.

When the sample is tiny, stray neutrons miss the sample and are not reflected onto the detector. This results in a resolution that is tighter than expected given the slit openings. If the sample width is available, we can use that to...
determine how much of the beam is intercepted by the sample, which we then use as an alternative second slit. This simple calculation isn’t quite correct for very low \( Q \), but data in this region will be contaminated by the direct beam, so we won’t be using those points.

When the sample is warped, it may act to either focus or spread the incident beam. Some samples are diffuse scatters, so we won’t be using those points.

For time of flight instruments, the wavelength dispersion is determined by the reduction process which usually bins the time channels in a way that sets a fixed relative resolution \( \Delta \lambda / \lambda \) for each bin.

Resolution in \( Q \) is computed from uncertainty in wavelength \( \sigma_\lambda \) and angle \( \sigma_\theta \) using propagation of errors:

\[
\sigma_Q^2 = \left( \frac{\partial Q}{\partial \lambda} \right)^2 \sigma_\lambda^2 + \left( \frac{\partial Q}{\partial \theta} \right)^2 \sigma_\theta^2 + 2 \left( \frac{\partial Q}{\partial \lambda} \frac{\partial Q}{\partial \theta} \right) \sigma_\lambda \sigma_\theta
\]

\[
Q = 4\pi \sin(\theta) / \lambda
\]

\[
\frac{\partial Q}{\partial \lambda} = -4\pi \sin(\theta) / \lambda^2 = -Q / \lambda
\]

\[
\frac{\partial Q}{\partial \theta} = 4\pi \cos(\theta) / \lambda = \cos(\theta) \cdot Q / \sin(\theta) = Q / \tan(\theta)
\]

With no correlation between wavelength dispersion and angular divergence, \( \sigma_\theta \lambda = 0 \), yielding the traditional form:

\[
\left( \frac{\Delta Q}{Q} \right)^2 = \left( \frac{\Delta \lambda}{\lambda} \right)^2 + \left( \frac{\Delta \theta}{\tan(\theta)} \right)^2
\]

Computationally, \( 1 / \tan(\theta) \to \infty \) at \( \theta = 0 \), so it is better to use the direct calculation:

\[
\Delta Q = 4\pi / \lambda \sqrt{\sin(\theta)^2 (\Delta \lambda / \lambda)^2 + \cos(\theta)^2 \Delta \theta^2}
\]

Wavelength dispersion \( \Delta \lambda / \lambda \) is usually constant (e.g., for AND/R it is 2% FWHM), but it can vary on time-of-flight instruments depending on how the data is binned.

Angular divergence \( \delta \theta \) comes primarily from the slit geometry, but can have broadening or focusing due to a warped sample. The FWHM divergence in radians due to slits is:

\[
\Delta \theta_{\text{slits}} = \frac{1}{2} \frac{s_1 + s_2}{d_1 - d_2}
\]

where \( s_1, s_2 \) are slit openings edge to edge and \( d_1, d_2 \) are the distances between the sample and the slits. For tiny samples of width \( m \), the sample itself can act as a slit. If \( s = m \sin(\theta) \) is smaller than \( s_2 \) for some \( \theta \), then use:

\[
\Delta \theta_{\text{slits}} = \frac{1}{2} \frac{s_1 + m \sin(\theta)}{d_1}
\]

The sample broadening can be read off a rocking curve using:

\[
\Delta \theta_{\text{sample}} = w - \Delta \theta_{\text{slits}}
\]

where \( w \) is the measured FWHM of the peak in degrees. Broadening can be negative for concave samples which have a focusing effect on the beam. This constant should be added to the computed \( \Delta \theta \) for all angles and slit geometries. You will not usually have this information on hand, but you can leave space for users to enter it if it is available.

FWHM can be converted to 1-\( \sigma \) resolution using the scale factor of \( 1 / \sqrt{8 \ln 2} \).

With opening slits we assume \( \Delta \theta / \theta \) is held constant, so if you know \( s \) and \( \theta_o \) at the start of the opening slits region you can compute \( \Delta \theta / \theta_o \), and later scale that to your particular \( \theta \):

\[
\Delta \theta(Q) = \Delta \theta / \theta_o \cdot \theta(Q)
\]

Because \( d \) is fixed, that means \( s_1(\theta) = s_1(\theta_o) \cdot \theta / \theta_o \) and \( s_2(\theta) = s_2(\theta_o) \cdot \theta / \theta_o \).
3.3.5 Applying Resolution

The instrument resolution is applied to the theory calculation on a point by point basis using a value of $\Delta Q$ derived from $\Delta \lambda$ and $\Delta \theta$. Assuming the resolution is well approximated by a Gaussian, convolve applies it to the calculated theory function.

The convolution at each point $k$ is computed from the piece-wise linear function $\bar{R}_i(q)$ defined by the reflectivity $R(Q_i)$ computed at points $Q_i \in Q_{\text{calc}}$

$$\bar{R}_i(q) = m_i q + b_i$$

$$m_i = (R_{i+1} - R_i)/(Q_{i+1} - Q_i)$$

$$b_i = R_i - m_i Q_i$$

and the Gaussian of width $\sigma_k = \Delta Q_k$

$$G_k(q) = \frac{1}{\sqrt{2\pi \sigma_k}} e^{(q-Q_k)^2/(2\sigma_k^2)}$$

using the piece-wise integral

$$\hat{R}_k = \sum_{i=i_{\text{min}}}^{i_{\text{max}}} \int_{Q_i}^{Q_{i+1}} \bar{R}_i(q) G_k(q) dq$$

The range $i_{\text{min}}$ to $i_{\text{max}}$ for point $k$ is defined to be the first $i$ such that $G_k(Q_i) < 0.001$, which is about $3\Delta Q_k$ away from $Q_k$.

By default the calculation points $Q_{\text{calc}}$ are the same nominal $Q$ points at which the reflectivity was measured. If the data was measured densely enough, then the piece-wise linear function $\bar{R}$ will be a good approximation to the underlying reflectivity. There are two places in particular where this assumption breaks down. One is near the critical edge for a sample that has sharp interfaces, where the reflectivity drops precipitously. The other is in thick samples, where the Kissig fringes are so close together that the instrument cannot resolve them separately.

The method Probe.critical_edge() fills in calculation points near the critical edge. Points are added linear around $Q_c$ for a range of $\pm \delta Q_c$. Thus, if the backing medium SLD or the theta offset are allowed to vary a little during the fit, the region after the critical edge may still be over-sampled. The method Probe.oversample() fills in calculation points around every point, giving each $\hat{R}$ a firm basis of support.

While the assumption of Gaussian resolution is reasonable on fixed wavelength instruments, it is less so on time of flight instruments, which have asymmetric wavelength distributions. You can explore the effects of different distributions by subclassing Probe and overriding the _apply_resolution method. We will happily accept code for improved resolution calculators and non-gaussian convolution.

3.3.6 Back reflectivity

While reflectivity is usually performed from the sample surface, there are many instances where them comes instead through the substrate. For example, when the sample is soaked in water or D$_2$O, a neutron beam will not penetrate well and it is better to measure the sample through the substrate. Rather than reversing the sample representation, these datasets can be flagged with the attribute back_reflectivity=True, and the sample constructed from substrate to surface as usual.

When the beam enters the side of the substrate, there is a small refractive shift in $Q$ based on the angle of the beam relative to the side of the substrate. The refracted beam reflects off the reversed film then exits the substrate on the other side, with an opposite refractive shift. Depending on the absorption coefficient of the substrate, the beam will be attenuated in the process.

The refractive shift and the reversing of the film are automatically handled by the underlying reflectivity calculation. You can even combine measurements through the sample surface and the substrate into a single measurement, with
negative $Q$ values representing the transition from surface to substrate. This is not uncommon with magnetic thin film samples.

Usually the absorption effects of the substrate are accounted for by measuring the incident beam through the same substrate before normalizing the reflectivity. There is a slight difference in path length through the substrate depending on angle, but it is not significant. When this is not the case, particularly for measurements which cross from the surface to substrate in the same scan, an additional back_absorption parameter can be used to scale the back reflectivity relative to the surface reflectivity. There is an overall intensity parameter which scales both the surface and the back reflectivity.

The interaction between back_reflectivity, back_absorption, sample representation and $Q$ value can be somewhat tricky. It

### 3.3.7 Alignment offset

It can sometimes be difficult to align the sample, particularly on X-ray instruments. Unfortunately, a misaligned sample can lead to an error in the measured position of the critical edge. Since the statistics for the measurement are very good in this region, the effects on the fit can be large. By representing the angle directly, an alignment offset can be incorporated into the reflectivity calculation. Furthermore, the uncertainty in the alignment can be estimated from the alignment scans, and this information incorporated directly into the fit. Without the theta offset correction you would need to compensate for the critical edge by allowing the scattering length density of the substrate to vary during the fit, but this would lead to incorrectly calculated reflectivity for the remaining points. For example, the simulation toffset.py shows more than 5% error in reflectivity for a silicon substrate with a 0.005° offset.

The method `Probe.alignment_uncertainty` computes the uncertainty in an alignment from the information in a rocking curve. The alignment itself comes from the peak position in the rocking curve, with uncertainty determined from the uncertainty in the peak position. Note that this is not the same as the width of the peak; the peak stays roughly the same width as statistics are improved, but the uncertainty in position and width will decrease. There is an additional uncertainty in alignment due to motor step size, easily computed from the variance in a uniform distribution. Combined, the uncertainty in theta_offset is:

$$\Delta \theta \approx \sqrt{\frac{w^2}{I} + \frac{d^2}{12}}$$

where $w$ is the full-width of the peak in radians at half maximum, $I$ is the integrated intensity under the peak and $d$ is the motor step size in radians.

### 3.3.8 Scattering Factors

The effective scattering length density of the material is dependent on the composition of the material and on the type and wavelength of the probe object. Using the chemical formula, `scattering_factors` computes the scattering factors ($\rho_s$, $\rho_t$, $\rho_{inc}$) associated with the material. This means the same sample representation can be used for X-ray and neutron experiments, with mass density as the fittable parameter. For energy dependent materials (e.g., Gd for neutrons), then scattering factors will be returned for all of the energies in the probe. (Note: energy dependent neutron scattering factors are not yet implemented in periodic table.)

The returned scattering factors are normalized to density=1 g·cm$^{-3}$. To use these values in the calculation of reflectivity, they need to be scaled by density and volume fraction. Using normalized density, the value returned by scattering_factors can be cached so only one lookup is necessary during the fit even when density is a fitting parameter.

The material itself can be flagged to use the incoherent scattering factor $\rho_{inc}$ which is by default ignored.

Magnetic scattering factors for the material are not presently available in the periodic table. Interested parties may consider extending periodic table with magnetic scattering information and adding support to PolarizedNeutronProbe.

---

3.4 Materials

Because this is elemental nickel, we already know it’s density. For compounds such as ‘SiO2’ we would have to specify an additional density=2.634 parameter.

Common materials defined in materialdb:

- air, water, silicon, sapphire, ...

Specific elements, molecules or mixtures can be added using the classes in refl1d.material:

- SLD unknown material with fittable SLD
- Material known chemical formula and fittable density
- Mixture known alloy or mixture with fittable fractions

3.5 Sample Representation

- Stacks
- Multilayers
- Interfaces
- Slabs
- Magnetic layers
- Polymer layers
- Functional layers
- Freeform layers
  - Comparison of models
  - Future work
- Subclassing Layer

3.5.1 Stacks

Reflectometry samples consist of 1-D stacks of layers joined by error function interfaces. The layers themselves may be uniform slabs, or the scattering density may vary with depth in the layer. The first layer in the stack is the substrate and the final layer is the surface. Surface and substrate are assumed to be semi-infinite, with any thickness ignored.

3.5.2 Multilayers

3.5.3 Interfaces

The interface between layers is assumed to smoothly follow and error function profile to blend the layer above with the layer below. The interface value is the 1-σ gaussian roughness. Adjacent flat layers with zero interface will act like a step function, while positive values will introduce blending between the layers.

Blending is usually done with the Nevot-Croce formalism, which scales the index of refraction between two layers by \( \exp(-2k_n k_{n+1} \sigma^2) \). We show both a step function profile for the interface, as well as the blended interface.

**Note:** The blended interface representation is limited to the neighbouring layers, and is not an accurate representation of the effective reflectivity profile when the interface value is large relative to the thickness of the layer.
We will have a mechanism to force the use of the blended profile for direct calculation of the interfaces rather than using the interface scale factor.

### 3.5.4 Slabs

Materials can be stacked as slabs, with a thickness for each layer and roughness at the top of each layer. Because this is such a common operation, there is special syntax to do it, using ‘|’ as the layer separator and () to specify thickness and interface. For example, the following is a 30 Å gold layer on top of silicon, with a silicon:gold interface of 5 Å and a gold:air interface of 2 Å:

```python
>> from refl1d import *
>> sample = silicon(0,5) | gold(30,2) | air
>> print sample
Si | Au(30) | air
```

Individual layers and stacks can be used in multiple models, with all parameters shared except those that are explicitly made separate. The syntax for doing so is similar to that for lists. For example, the following defines two samples, one with Si+Au/30+air and the other with Si+Au/30+alkanethiol/10+air, with the silicon/gold layers shared:

```python
>> alkane_thiol = Material('C2H4OHS',bulk_density=0.8,name='thiol')
>> sample1 = silicon(0,5) | gold(30,2) | air
>> sample2 = sample1[:-1] | alkane_thiol(10,3) | air
>> print sample2
Si | Au(30) | thiol(10) | air
```

Stacks can be repeated using a simple multiply operation. For example, the following gives a cobalt/copper multilayer on silicon:

```python
>> Cu = Material('Cu')
>> Co = Material('Co')
>> sample = Si | [Co(30) | Cu(10)]*20 | Co(30) | air
>> print sample
Si | [Co(30) | Cu(10)]*20 | Co(30) | air
```

Multiple repeat sections can be included, and repeats can contain repeats. Even freeform layers can be repeated. By default the interface between the repeats is the same as the interface between the repeats and the cap. The cap interface can be set explicitly. See model.Repeat for details.

### 3.5.5 Magnetic layers

### 3.5.6 Polymer layers

### 3.5.7 Functional layers

### 3.5.8 Freeform layers

Freeform profiles allow us to adjust the shape of the depth profile using control parameters. The profile can directly represent the scattering length density as a function of depth (a FreeLayer), or the relative fraction of one material and another (a FreeInterface). With a freeform interface you can simultaneously fit two systems which should share the same volume profile but whose materials have different scattering length densities. For example, a polymer in deuterated and undeuterated solvents can be simultaneously fit with freeform profiles.

We have multiple representations for freeform profiles, each with its own strengths and weaknesses:

- monotone cubic interpolation (`reflld.mono`)
• parameteric B-splines \texttt{(ref1d.freeform)}

• Chebyshev interpolating polynomials \texttt{(ref1d.cheby)}

At present, monotone cubic interpolation is the most developed, but work on all representations is in flux. In particular not every representation supports all features, and the programming interface may vary. See the documentation for the individual models for details.

\subsection*{Comparison of models}

There are a number of issues surrounding the choice of model.

• How easy is it to bound the profile values
  
  If the you can put reasonable bounds on the control points, then the user can bring to bear prior information to limit the search space. For example, it is common to add an unknown silicon-oxide profile to the surface of silicon, with SLD varying between the values for Si and SiO$_2$.

• How easy is it to edit the profile interactively
  
  Given a representation of the freeform layer, we want to be able to plot control points that you can drag in order to change the shape of the profile.

• Is the profile stable or does it oscillate wildly
  
  Many systems are best described by smoothly varying density profiles. If the profile oscillates wildly it makes the search for optimal parameters more difficult.

• Can you change the order of interpolation and preserve the profile
  
  While the current code does not support it, we would like to be able to select the freeform profile order automatically, using the minimum order we can to achieve $\chi^2 = 1$, and rejecting profiles which overfit the data. For now this is done by hand, performing fits with different orders independently, but there are likely to be speed gains by first fitting coarse models with low Q then adding detail to the profile while adding additional Q values.

• Is the representation unique? Are the control parameters strongly correlated?
  
  Fitting and uncertainty analysis benefit from unique solutions. If the model representation is matched by a family of parameters it is more difficult to interpret the results of the uncertainty analysis or to get convergence from the parameter refinement engine.

Monotone cubic interpolation is the easiest to control. The value of the interpolating polynomial lies mostly within the range of the control points, and the profile goes through the control points. This means you can set up bounds on the control parameters that limit the profile to a certain range of scattering length densities in a region of the profile. It also leads to a very intuitive interactive profile editor since the control points can be moved directly on profile view. However, although the profile is $C^1$ smooth everywhere, the $C^2$ transitions can be abrupt at the control points. Better algorithms for selecting the gradient exist but have not been implemented, so this may improve in the future.

Parametric B-splines are commonly used in computer graphics because they create pleasing curves. The interpolating polynomial lies within the convex hull of the control points. Unfortunately the distance between the curve and the control point can be large, and this makes it difficult to set reasonable bounds on the values of the control points. One can reformulate the interpolation so that control points lie on the curve and still preserve the property of pleasing curves, but this can lead to wild oscillations in the profile when the control points become too close together. While the natural representation can be used in an interactive profile editor, the fact that the control points are sometimes far away from the profile makes this inconvenient. The complementary representation is used in programs such as Microsoft Excel, with the control point directly on the curve and a secondary control point to adjust the slope at that control point.

Chebyshev interpolating polynomials are a near optimal representation for a function over an interval with respect to the maximum norm. The interpolating polynomial is a weighted sum $\sigma^n_{i=0} c_i T_i(z)$ of the Chebyshev basis polynomials $T_i$ with Chebyshev coefficients $c_i$. One very interesting property is that the lower order coefficients remain the same.
has higher order interpolation polynomials are constructed. This makes the Chebyshev polynomials very interesting candidates for a freeform profile fitter which selects the order of the profile as part of the fit. Chebyshev interpolating polynomials can exhibit wild oscillations if the coefficients become large, so the smoothness can be somewhat controlled by limiting these higher values, but we have not explored this in depth. The Chebyshev coefficient values are not directly tied to the profile, so there is no intuitive way to directly control the coefficients in an interactive editor. The complementary representation uses the profile value at the chebyshev nodes for specific positions $z_i$ on the profile. This representation is much more natural for an interactive editor, but some choices of control values will lead to wild oscillations between the nodes. Similarly the complementary representation is unsuitable as a representation for the fittable parameters since the bounds on the parameters do not directly limit the range of possible values of the profile.

Future work

We only have polynomial spline representations for our profiles. Similar profiles could be constructed from different basis functions such as wavelets, the idea being to find a multiscale representation of your profile and use model selection techniques to determine the most coarse grained representation that matches your data.

Totally freeform representations as separately controlled microslab heights would also be interesting in the context of a maximum entropy fitting engine: find the smoothest profile which matches the data, for some definition of ‘smooth’. Some possible smoothness measures are the mean squared distance from zero, the number of sign changes in the second derivative, the sum of the absolute value of the first derivative, the maximum flat region, the minimum number of flat slabs, etc. Given that reflectometry inversion is not unique, the smoothness measure must correspond to the likelihood of finding the system in that particularly state: that is, don’t expect your sample to show zebra stripes unless you are on an African safari or visiting a zoo.

3.5.9 Subclassing Layer

3.6 Experiment

- Direct Calculation

The Experiment object links a sample with an experimental probe. The probe defines the Q values and the resolution of the individual measurements, and returns the scattering factors associated with the different materials in the sample.

For the simple case of exploring the reflectivity of new samples, this means that you must define the purposes:

- defining the instrument resolution
- providing the scattering factors for materials

Because our models allow representation based on composition, it is no longer trivial to compute the reflectivity from the model. We now have to look up the effective scattering density based on the probe type and probe energy. You’ve already seen this in Subclassing Layer: the render method for the layer requires the probe to look up the material scattering factors.

3.6.1 Direct Calculation

Rather than using Stack <refl1d.model.Stack, Probe and class:Experiment <refl1d.experiment.Experiment, we can compute reflectivities directly with the functions in
refl1d.reflectivity. These routines provide the raw calculation engines for the optical matrix formalism, converting microslab models of the sample into complex reflectivity amplitudes, and convolving the resulting reflectivity with the instrument resolution.

The following performs a complete calculation for a silicon substrate with 5 Å roughness using neutrons. The theory is sampled at intervals of 0.001, which is convolved with a 1% $\Delta Q/Q$ resolution function to yield reflectivities at intervals of 0.01.

```python
>>> from numpy import arange
>>> from refl1d.reflectivity import reflectivity_amplitude as reflamp
>>> from refl1d.reflectivity import convolve
>>> Qin = arange(0,0.21,0.001)
>>> w,rho,irho,sigma = zip((0,2.07,0,5),(0,0,0,0))
>>> # the last layer has no interface
>>> r = reflamp(kz=Qin/2, depth=w, rho=rho, irho=irho, sigma=sigma[:-1])
>>> Rin = (r*r.conj()).real
>>> Q = arange(0,0.2,0.01)
>>> dQ = Q*0.01 # resolution $dQ/Q = 0.01$
>>> R = convolve(Qin, Rin, Q, dQ)
>>> print("\n".join("Q: %.2f R: %.5e"%(Qi,Ri) for Qi,Ri in zip(Q,R)))
Q: 0.00 R: 1.00000e+00
Q: 0.01 R: 3.11332e-02
Q: 0.02 R: 3.30684e-03
...  
Q: 0.19 R: 2.10084e-07
```

### 3.7 Fitting

Obtaining a good fit depends foremost on having the correct model to fit.

Too many layers, too few layers, too limited fit ranges, too open fit ranges, all of these can make fitting difficult. For example, forgetting the SiOx layer on the silicon substrate will distort the model of a polymer film.

Even with the correct model, there are systematic errors to address (see _data_guide). A warped sample can lead to broader resolution than expected near the critical edge, and `sample_broadening=value` must be specified when loading the data. Small errors in alignment of the sample or the slits will move the measured critical edge, and so `probe.theta_offset` may need to be fitted. Points near the critical edge are difficult to compute correctly with resolution because the reflectivity varies so quickly. Using `refl1d.probe.Probe.critical_edge()`, the density of the points used to compute the resolution near the critical edge can be increased. For thick samples the resolution will integrate over multiple Kissig fringes, and `refl1d.probe.Probe.over_sample()` will be needed to average across them and avoid aliasing effects.
3.7.1 Quick Fit

While generating an appropriate model, you will want to perform a number of quick fits. The Nelder-Mead simplex algorithm (fit=amoeba) works well for this. You will want to run it with steps between 1000 and 3000 so the algorithm has a chance to converge. Restarting a number of times (somewhere between 3 and 100) gives a reasonably thorough search of the fit space. From the graphical user interface (refl_gui), using starts=1 and clicking the fit button to improve the fit as needed works pretty well. From the command line interface (refl_cli), the command line will be something like:

```
refl1d --fit=amoeba --steps=1000 --starts=20 --parallel model.py --store=T1
```

The command line result can be improved by using the previous fit value as the starting point for the next fit:

```
refl1d --fit=amoeba --steps=1000 --starts=20 --parallel model.py --store=T1 --pars=T1/model.par
```

Differential evolution (fit=de) and random lines (fit=rl) are alternatives to amoeba, perhaps a little more likely to find the global minimum but somewhat slower. These are population based algorithms in which several points from the current population are selected, and based on their position and value, a new point is generated. The population is specified as a multiplier on the number of parameters in the model, so for example an 8 parameter model with DE’s default population (pop=10) would create 80 points each generation. Random lines with a large population is fast but is not good at finding isolated minima away from the general trend, so its population defaults to pop=0.5. These algorithms can be called from the command line as follows:

```
refl1d --fit=de --steps=3000 --parallel model.py --store=T1
refl1d --fit=rl --steps=3000 --starts=200 --reset --parallel model.py --store=T1
```

Of course, –pars can be used to start from a previously completed fit.

3.7.2 Uncertainty Analysis

More important than the optimal value of the parameters is an estimate of the uncertainty in those values. By casting our problem as the likelihood of seeing the data given the model, we not only give ourselves the ability to incorporate prior information into the fit systematically, but we also give ourselves a strong foundation for assessing the uncertainty of the parameters.

Uncertainty analysis is performed using DREAM (fit=dream). This is a Markov chain Monte Carlo (MCMC) method with a differential evolution step generator. Like simulated annealing, the MCMC explores the space using a random walk, always accepting a better point, but sometimes accepting a worse point depending on how much worse it is.

DREAM can be started with a variety of initial populations. The random population (init=random) distributes the initial points using a uniform distribution across the space of the parameters. Latin hypersquares (init=lhs) improves on random by making sure that there is on value for each subrange of every variable. The covariance population (init= cov) selects points from the uncertainty ellipse computed from the derivative at the initial point. This method will fail if the fitting parameters are highly correlated and the covariance matrix is singular. The epsilon ball population (init=eps) starts DREAM from a tiny region near the initial point and lets it expand from there. It can be useful to start with an epsilon ball from the previous best point when DREAM fails to converge using a more diverse initial population.

The Markov chain will take time to converge on a stable population. This burn in time needs to be specified at the start of the analysis. After burn, DREAM will collect all points visited for N iterations of the algorithm. If the burn time was long enough, the resulting points can be used to estimate uncertainty on parameters.

A common command line for running DREAM is:

```
refl1d --fit=dream --burn=1000 --steps=1000 --init= cov --parallel --pars=T1/model.par model.py --store=T2
```

The file T1/model.err contains a table showing for each parameter the mean(std), median and best values, and the 68% and 95% credible intervals. The mean and standard deviation are computed from all the samples in the returned
distribution. These statistics are not robust: if the Markov process has not yet converged, then outliers will significantly distort the reported values. Standard deviation is reported in compact notation, with the two digits in parentheses representing uncertainty in the last two digits of the mean. Thus, for example, 24.9(28) is 24.9 ± 2.8. Median is the best value in the distribution. Best is the best value ever seen. The 68% and 95% intervals are the shortest intervals that contain 68% and 95% of the points respectively. In order to report 2 digits of precision on the 95% interval, approximately 1000000 draws from the distribution are required, or steps = 1000000/(#parameters #pop). The 68% interval will require fewer draws, though how many has not yet been determined.

Histogramming the set of points visited will gives a picture of the probability density function for each parameter. This histogram is generated automatically and saved in T1/model-var.png. The histogram range represents the 95% credible interval, and the shaded region represents the 68% credible interval. The green line shows the highest probability observed given that the parameter value is restricted to that bin of the histogram. With enough samples, this will correspond to the maximum likelihood value of the function given that one parameter is restricted to that bin. In practice, the analysis has converged when the green line follows the general shape of the histogram.

The correlation plots show that the parameters are not uniquely determined from the data. For example, the thickness of lamellae 3 and 4 are strongly anti-correlated, yielding a 95% CI of about 1 nm for each compared to the bulk nafion thickness CI of 0.2 nm. Summing lamellae thickness in the sampled points, we see the overall lamellae thickness has a CI of about 0.3 nm. The correlation plot is saved in T1/model-corr.png.

To assure ourselves that the uncertainties produced by DREAM do indeed correspond to the underlying uncertainty in the model, we perform a Monte Carlo forward uncertainty analysis by selecting 50 samples from the computed
posterior distribution, computing the corresponding reflectivity and calculating the normalized residuals. Assuming that our measurement uncertainties are approximately normally distributed, approximately 68% of the normalized residuals should be within +/- 1 of the residual for the best model, and 98% should be within +/- 2. Note that our best fit does not capture all the details of the data, and the underlying systematic bias is not included in the uncertainty estimates.

Plotting the profiles generated from the above sampling method, aligning them such that the cross correlation with the best profile is maximized, we see that the precise details of the lamellae are uncertain but the total thickness of the lamellae structure is well determined. Bayesian analysis can also be used to determine relative likelihood of different number of layers, but we have not yet performed this analysis. This plot is stored in T1/model-errors.png.

The trace plot, T1/model-trace.png, shows the mixing properties of the first fitting parameter. If the Markov process is well behaved, the trace plot will show a lot of mixing. If it is ill behaved, and each chain is stuck in its own separate local minimum, then distinct lines will be visible in this plot.

The convergence plot, T1/model-logp.png, shows the log likelihood values for each member of the population. When the Markov process has converged, this plot will be flat with no distinct lines visible. If it shows a general upward sweep, then the burn time was not sufficient, and the analysis should be restarted. The ability to continue to burn from the current population is not yet implemented.

Given sufficient burn time, points in the search space will be visited with probability proportional to the goodness of fit. It can be difficult to determine the correct amount of burn time in advance. If burn is not long enough, then the population of log likelihood values will show an upward sweep. Similarly, if steps is insufficient, the likelihood observed as a function of parameter value will be sparsely sampled, and the maximum likelihood curve will not match the posterior probability histogram. To correct these issues, the DREAM analysis can be extended using the –resume option. Assume the previous run completed with Markov chain convergence achieved at step 500. The following command line will generate an additional 600 steps so that the posterior sample size is 1600, then run an additional 500 steps of burn to remove the initial upward sweep in the log likelihood plot:

```
refl1d --fit=dream --burn=500 --steps=1600 --parallel --resume=T2 --store=T3
```

The results are stored in directory T3.

Just because all the plots are well behaved does not mean that the Markov process has converged on the best result. It is practically impossible to rule out a deep minimum with a narrow acceptance region in an otherwise unpromising part of the search space.

In order to assess the DREAM algorithm for suitability for reflectometry fitting we did a number of tests. Given that the fit surface is multimodal, we need to know that the uncertainty analysis can return multiple modes. Because the fit problems may also be ill-conditioned, with strong correlations or anti-correlations between some parameters, the uncertainty analysis needs to be able to correctly indicate that the correlations exist. Simple Metropolis-Hastings sampling does not work well in these conditions, but DREAM is able to handle them.

### 3.7.3 Using the posterior distribution

You can load the DREAM output population an perform uncertainty analysis operations after the fact:

```
$ ipython -pylab
import dream.state
state = dream.state.load_state(modelname)
state.mark_outliers() # ignore outlier chains
state.show()  # Plot statistics
```

You can restrict a variable to a certain range when doing plots. For example, to restrict the third parameter to [0.8-1.0] and the fifth to [0.2-0.4]:

```
refl1d --fit=dream --burn=500 --steps=1600 --parallel --resume=T2 --store=T3
```
You can also add derived variables using a function to generate the derived variable. For example, to add a parameter which is \( p[0]+p[1] \) use:

```python
define vars(lambda: p[0]+p[1], labels=['x+y'])
```

You can generate multiple derived parameters at a time with a function that returns a sequence:

```python
define vars(lambda: (p[0]*p[1],p[0]-p[1]), labels=['x*y','x-y'])
```

These new parameters will show up in your plots:

```python
state.show()
```

The plotting code is somewhat complicated, and matplotlib doesn’t have a good way of changing plots interactively. If you are running directly from the source tree, you can modify the dream plotting libraries as you need for a one-off plot, the replot the graph:

```python
# ... change the plotting code in dream.views/dream_corrplot
reload(dream.views)
reload(dream.corrplot)
state.show()
```

Be sure to restore the original versions when you are done. If the change is so good that everyone should use it, be sure to feed it back to the community via https://github.com/reflectometry/refl1d.

### 3.7.4 Reporting results

As with any parametric modeling technique, you cannot say that the model is correct and has certain parameter value, only that the observed data is consistent with the model and the given parameter values. There may be other models within the parameter search space that are equally consistent, but which were not discovered by Refl1D, particularly if you are forced to use –init=eps to achieve convergence. This is true even for models which exhibit good convergence:

- the marginal maximum likelihood (the green line) follows the marginal probability density (the blue line)
- the log likelihood function is flat, not sweeping upward
- the individual parameter traces exhibit good mixing
- the marginal probability density is unimodal and roughly normal
- the joint probabilities show no correlation structure
- \( \chi^2 \approx 1 \)
- the residuals plot shows no structure

The following blurb can be used as a description of the analysis method when reporting your results:

Refl1D[1] was used to model the reflectivity data. The sample depth profile is represented as a series of slabs of varying scattering length density and thickness with gaussian interfaces between them. Freeform sections of the profile are modeled using monotonic splines. Reflectivity is computed using the Abeles optical matrix method, with interfacial effects computed by the method of Nevot and Croce or by approximating the interfaces by a series of thin slabs. Refl1d supports simultaneous refinement of multiple reflectivity data sets with constraints between the models.
Refl1D uses a Bayesian approach to determine the uncertainty in the model parameters. By representing the problem as the likelihood of observing the measured reflectivity curve given a particular choice of parameters, Refl1D can use Markov Chain Monte Carlo (MCMC) methods[2] to draw a random sample from the joint parameter probability distribution. This sample can then be used to estimate the probability distribution for each individual parameter.


If you are reporting maximum likelihood and credible intervals:

The parameter values reported are those from the model which best fits the data, with uncertainty determined from the range of parameter values which covers 68% of the sample set. This corresponds to the $1-\sigma$ uncertainty level if the sample set were normally distributed.

If you are reporting mean and standard deviation:

The reported parameter values are computed from the mean and standard deviation of the sample set. This corresponds to the best fitting normal distribution to marginal probability distribution for the parameter.

There are caveats to reporting mean and standard deviation. The technique is not robust. If burn-in is insufficient, if the distribution is multi-modal, or if the distribution has long tails, then the reported mean may correspond to a bad fit, and the standard deviation can be huge. [We should confirm this by modeling a cauchy distribution]

### 3.7.5 Publication Graphics

The matplotlib package is capable of producing publication quality graphics for your models and fit results, but it requires you to write scripts to get the control that you need. These scripts can be run from the refl1d application by first loading the model and the fit results then accessing their data directly to produce the plots that you need.

The model file (call it plot.py) will start with the following:

```python
import sys
from refl1d.cli import load_problem, load_best

model, store = sys.argv[1:3]

problem = load_problem([model])
load_best(problem, os.path.join(store, model[:-3]+".par"))
chisq = problem.chisq

print "chisq",chisq
```

Assuming your model script is in model.py and you have run a fit with --store=X5, you can run this file using:

```
$ refl1d plot.py model.py X5
```

Now model.py is loaded and the best fit parameters are set.

To produce plots, you will need access to the data and the theory. This can be complex depending on how many models you are fitting and how many datasets there are per model. For refl1d.fitproblem.FitProblem models, the refl1d.experiment.Experiment object is referenced by `problem.fitness`. For refl1d.fitproblem.MultiFitProblem models, you need to use `problem.models[k].fitness` to access the experiment for model k. Profiles and reflectivity theory are returned from methods in experiment. The refl1d.probe.Probe data for the experiment is referenced by `experiment.probe`. This will
have attributes for $Q$, $dQ$, $R$, $dR$, $T$, $dT$, and $L$, $dL$, as well as methods for plotting the data. This is not quite so simple: the sample may be non uniform, and composed of multiple samples for the same probe, and at the same time the probe may be composed of independent measurements kept separate so that you can fit alignment angle and overall intensity. Magnetism adds another level of complexity, with extra profiles associated with each sample and separate reflectivities for the different spin states.

How does this work in practice? Consider a simple model such as nifilm-fit from the example directory. We can access the parts by extending plot.py as follows:

```python
experiment = problem.fitness
z,rho,irho = experiment.smooth_profile(dz=0.2)
# ... insert profile plotting code here ...
QR = experiment.reflectivity()
for p,th in self.parts(QR):
    # ... insert reflectivity plotting code here ...

Next we can reload the the error sample data from the DREAM MCMC sequence:

```python
from bumps.dream.state import load_state
from bumps.errplot import calc_errors_from_state
from refl1d.errors import align_profiles

state = load_state(os.path.join(store, model[:-3]))
state.mark_outliers()
# ... insert correlation plots, etc. here ...
profiles,slabs,Q,residuals = calc_errors_from_state(problem, state)
aligned_profiles = align_profiles(profiles, slabs, 2.5)
# ... insert profile and residuals uncertainty plots here ...
```

The function `refl1d.errors.calc_errors()` provides details on the data structures for `profiles`, $Q$ and residuals. Look at the source in `refl1d/errors.py` to see how this data is used to produce the error plots with `_profiles_overplot, _profiles_contour, _residuals_overplot and _residuals_contour`. The source is available from:

https://github.com/reflectometry/refl1d

Putting the pieces together, here is a skeleton for a specialized plotting script:

```python
import sys
import pylab
import dream.state
from bumps.cli import load_problem, load_best
from refl1d.errors import calc_errors_from_state, align_profiles

model, store = sys.argv[1:3]

problem = load_problem([model])
load_best(problem, os.path.join(store, model[:-3]+".par"))

chisq = problem.chisq
experiment = problem.fitness
z,rho,irho = experiment.smooth_profile(dz=0.2)
# ... insert profile plotting code here ...
QR = experiment.reflectivity()
for p,th in self.parts(QR):
    # ... insert reflectivity plotting code here ...

if 1: # Loading errors is expensive; may not want to do so all the time.
    state = dream.state.load_state(os.path.join(store, model[:-3]))
```

3.7. Fitting
For the common problem of generating profile error plots aligned on a particular interface, you can use the simpler align.py model:

```python
from refl1d.names import * align_errors(model='', store='', align='auto')
```

If you are using the command line then you should be able to type the following at the command prompt to generate the plots:

```
$ refl1d align.py <model>.py <store> [<align>] [1|2|n]
```

If you are using the GUI, you will have to set model, store and align directly in align.py each time you run.

Align is either auto for the current behaviour, or it is an interface number. You can align on the center of a layer by adding 0.5 to the interface number. You can count interfaces from the surface by prefixing with R. For example, 0 is the substrate interface, R1 is the surface interface, 2.5 is the the middle of layer 2 above the substrate.

You can plot the profiles and residuals on one plot by setting plots to 1, on two separate plots by setting plots to 2, or each curve on its own plot by setting plots to n. Output is saved in `<store>/<model>-err#.png`.

### 3.7.6 Tough Problems

With the toughest fits, for example freeform models with many control points, parallel tempering (fit=pt) is the most promising algorithm. This implementation is an extension of DREAM. Whereas DREAM runs with a constant temperature, T=1, parallel tempering runs with multiple temperatures concurrently. The high temperature points are able to walk up steep hills in the search space, possibly crossing over into a neighbouring valley. The low temperature points aggressively seek the nearest local minimum, rejecting any proposed point that is worse than the current. Differential evolution helps adapt the steps to the shape of the search space, increasing the chances that the random step will be a step in the right direction. The current implementation uses a fixed set of temperatures defaulting to Tmin=0.1 through Tmax=10 in nT=25 steps; future versions should adapt the temperature based on the fitting problem.

Parallel tempering is run like dream, but with optional temperature controls:

```
refl1d --fit=dream --burn=1000 --steps=1000 --init=cov --parallel --pars=T1/model.par model.py --store=T2
```

Parallel tempering does not yet generate the uncertainty plots provided by DREAM. The state is retained along the temperature for each point, but the code to generate histograms from points weighted by inverse temperature has not yet been written.

### 3.7.7 Command Line

The GUI version is slower because it frequently updates the graphs showing the best current fit.

Run multiple models overnight, starting one after the last is complete by creating a batch file (e.g., run.bat) with one line per model. Append the parameter –batch to the end of the command lines so the program doesn’t stop to show interactive graphs. You can view the fitted results in the GUI using:

```
refl1d --edit model.py --pars=T1/model.par
```
3.7.8 Other optimizers

There are several other optimizers that are included but aren’t frequently used.

BFGS (fit=newton) is a quasi-newton optimizer relying on numerical derivatives to find the nearest local minimum. Because the reflectometry problem often has correlated parameters, the resulting matrices can be ill-conditioned and the fit isn’t robust.

Particle swarm optimization (fit=ps) is another population based algorithm, but it does not appear to perform well for high dimensional problem spaces that frequently occur in reflectivity.

SNOBFIT (fit=snobfit) attempts to construct a locally quadratic model of the entire search space. While promising because it can begin to offer some guarantees that the search is complete given reasonable assumptions about the fitting surface, initial trials did not perform well and the algorithm has not yet been tuned to the reflectivity problem.

3.7.9 References


4.1 abeles - Pure python reflectivity calculator

```
calc
refl
```
Reflectometry as a function of kz for a set of slabs.

Optical matrix form of the reflectivity calculation.
O.S. Heavens, Optical Properties of Thin Solid Films

This is a pure python implementation of reflectometry provided for convenience when a compiler is not available. The refl1d application uses reflmodule to compute reflectivity.

```
refl1d.abeles.calc(kz, depth, rho, irho, sigma)
refl1d.abeles.refl(kz, depth, rho, irho=0, sigma=0, rho_index=None)
```
Reflectometry as a function of kz for a set of slabs.

**Parameters**

- **kz** [float[n] | Å⁻¹] Scattering vector $2\pi \sin(\theta)/\lambda$. This is $\frac{1}{2} Q_z$.
- **depth** [float[m] | Å] thickness of each layer. The thickness of the incident medium and substrate are ignored.
- **rho, irho** [float[n,k] | 10⁻⁶ Å⁻²] real and imaginary scattering length density for each layer for each kz Note: absorption cross section $\mu = 2\ irho/\lambda$
- **sigma** [float[m-1] | Å] interfacial roughness. This is the roughness between a layer and the subsequent layer. There is no interface associated with the substrate. The sigma array should have at least m-1 entries, though it may have m with the last entry ignored.
- **rho_index** [int[m]] index into rho vector for each kz

Slabs are ordered with the surface SLD at index 0 and substrate at index -1, or reversed if kz < 0.

4.2 cheby - Freeform - Chebyshev model

```
ChebyVF    Material in a solvent
FreeformCheby     A freeform section of the sample modeled with Chebyshev polynomials.
```
Freeform modeling with Chebyshev polynomials

Chebyshev polynomials $T_k$ form a basis set for functions over $[-1, 1]$. The truncated interpolating polynomial $P_n$ is a weighted sum of Chebyshev polynomials up to degree $n$:

$$f(x) \approx P_n(x) = \sum_{k=0}^{n} c_k T_k(x)$$

The interpolating polynomial exactly matches $f(x)$ at the chebyshev nodes $z_k$ and is near the optimal polynomial approximation to $f$ of degree $n$ under the maximum norm. For well behaved functions, the coefficients $c_k$ decrease rapidly, and furthermore are independent of the degree $n$ of the polynomial.

FreeformCheby models the scattering length density profile of the material within a layer, and ChebyVF models the volume fraction profile of two materials mixed in the layer.

The models can either be defined directly in terms of the Chebyshev coefficients $c_k$ with method = ‘direct’, or in terms of control points $(z_k, f(z_k))$ at the Chebyshev nodes cheby_points() with method = ‘interp’. Bounds on the parameters are easier to control using ‘interp’, but the function may oscillate wildly outside the bounds. Bounds on the oscillation are easier to control using ‘direct’, but the shape of the profile is difficult to control.

class refl1d.cheby.ChebyVF (thickness=0, interface=0, material=None, solvent=None, vf=None, name='ChebyVF', method='interp')

Bases: refl1d.model.Layer

Material in a solvent

Parameters

thickness [float | Angstrom] the thickness of the solvent layer
interface [float | Angstrom] the rms roughness of the solvent surface
material [Material] the material of interest
solvent [Material] the solvent or vacuum
vf [[float]] the control points for volume fraction

method = ‘interp’ [string | ‘direct’ or ‘interp’] freeform profile method

method is ‘direct’ if the vf values refer to chebyshev polynomial coefficients or ‘interp’ if vf values refer to control points located at $z_k$.

The control point $k$ is located at $z_k \in [0, L]$ for layer thickness $L$, as returned by cheby_points() called with n=len(vf) and range=[0, L].

The materials can either use the scattering length density directly, such as PDMS = SLD(0.063, 0.00006) or they can use chemical composition and material density such as PDMS=Material("C2H6OSi",density=0.965).

These parameters combine in the following profile formula:

$$sld(z) = material.sld \times \text{profile}(z) + solvent.sld \times (1 - \text{profile}(z))$$

constraints ()

Constraints

find(z)

Find the layer at depth $z$.

Returns layer, start, end

interface = None

ismagnetic
layer_parameters()
magnetism
parameters()
penalty()
Return a penalty value associated with the layer. This should be zero if the parameters are valid, and increasing as the parameters become more invalid. For example, if total volume fraction exceeds unity, then the penalty would be the amount by which it exceeds unity, or if z values must be sorted, then penalty would be the amount by which they are unsorted.

Note that penalties are handled separately from any probability of seeing a combination of layer parameters; the final solution to the problem should not include any penalized points.

render (probe, slabs)

thickness = None

class refl1d.cheby.FreeformCheby (thickness=0, interface=0, rho=[], irho=[], name='Cheby', method='interp')

Bases: refl1d.model.Layer

A freeform section of the sample modeled with Chebyshev polynomials.

sld (rho) and imaginary sld (irho) can be modeled with a separate polynomial orders.

constraints()
Constraints

find (z)
Find the layer at depth z.

Returns layer, start, end

interface = None

ismagnetic

layer_parameters()
magnetism
parameters()

Return parameters used to define layer

penalty()
Return a penalty value associated with the layer. This should be zero if the parameters are valid, and increasing as the parameters become more invalid. For example, if total volume fraction exceeds unity, then the penalty would be the amount by which it exceeds unity, or if z values must be sorted, then penalty would be the amount by which they are unsorted.

Note that penalties are handled separately from any probability of seeing a combination of layer parameters; the final solution to the problem should not include any penalized points.

render (probe, slabs)

Render slabs for use with the given probe

thickness = None

4.3 dist - Non-uniform samples
Inhomogeneous samples

In the presence of samples with short range order on scale of the coherence length of the probe in the plane, but long range disorder following some distribution of parameter values, the reflectivity can be computed from a weighted incoherent sum of the reflectivities for different values of the parameter.

DistributionExperiment allows the model to be computed for a single varying parameter. Multi-parameter dispersion models are not available.

class refl1d.dist.DistributionExperiment (experiment=None, P=None, distribution=None, coherent=False)

Bases: refl1d.experiment.ExperimentBase

Compute reflectivity from a non-uniform sample.

The parameter \( P \) takes on the values from \( \text{distribution} \) in the context of \( \text{experiment} \). Clearly, \( P \) should not be a fitted parameter, but the remaining experiment parameters can be fitted, as can the parameters of the distribution.

If \( \text{coherent} \) is true, then the reflectivity of the mixture is computed from the coherent sum rather than the incoherent sum.

See \( \text{Weights} \) for a description of how to set up the distribution.

format_parameters ()

is_reset ()

Returns True if a model reset was triggered.

name

nllf ()

Return the \(-\log(P(\text{data|model}))\).

Using the assumption that data uncertainty is uncorrelated, with measurements normally distributed with mean \( R \) and variance \( dR^2 \), this is just \( \text{sum( resid^2/2 + log(2*pi*dR^2)/2 )} \).

The current version drops the constant term, \( \text{sum(log(2*pi*dR^2)/2)} \).

numpoints ()

parameters ()

plot (plot_shift=None, profile_shift=None, view=None)

plot_profile ()

plot_reflectivity (show_resolution=False, view=None, plot_shift=None)

plot_weights ()

reflectivity (resolution=True)

residuals ()

restore_data ()

Restore original data after resynthesis.

resynth_data ()

Resynthesize data with noise from the uncertainty estimates.

save (basename)
save_profile (basename)
save_refl (basename)
simulate_data (noise=2)
    Simulate a random data set for the model
    Parameters:
    noise = 2 [float | %] Percentage noise to add to the data.

smooth_profile (dz=1)
    Compute a density profile for the material

step_profile ()
    Compute a scattering length density profile

update ()
    Called when any parameter in the model is changed.
    This signals that the entire model needs to be recalculated.

update_composition ()
    When the model composition has changed, we need to lookup the scattering factors for the new model.
    This is only needed when an existing chemical formula is modified; new and deleted formulas will be
    handled automatically.

write_data (filename, **kw)
    Save simulated data to a file

class refl1d.dist.Weights (edges=None, cdf=None, args=[], loc=None, scale=None, truncated=True)
Bases: object

Parameterized distribution for use in DistributionExperiment.

To support non-uniform experiments, we must bin the possible values for the parameter and compute the theory
function for one parameter value per bin. The weighted sum of the resulting theory functions is the value that
we compare to the data.

Performing this analysis requires a cumulative density function which can return the integrated value of the
probability density from -inf to x. The total density in each bin is then the difference between the cumulative
densities at the edges. If the distribution is wider than the range, then the tails need to be truncated and the bins
reweighted to a total density of 1, or the tail density can be added to the first and last bins. Weights of zero are
not returned. Note that if the tails are truncated, this may result in no weights being returned.

The vector edges contains the bin edges for the distribution. The function cdf returns the cumulative den-
sity function at the edges. The cdf function must implement the scipy.stats interface, with function signature
f(x,a1,a2,...,loc=0,scale=1). The list args defines the arguments a1, a2, etc. The underlying parameters are avail-
able as args[i]. Similarly, loc and scale define the distribution center and width. Use truncated=False if you
want the distribution tails to be included in the weights.

SciPy distribution D is used by specifying cdf=scipy.stats.D.cdf. Useful distributions include:

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>norm</td>
<td>Gaussian distribution.</td>
</tr>
<tr>
<td>halfnorm</td>
<td>Right half of a gaussian.</td>
</tr>
<tr>
<td>triang</td>
<td>Triangle distribution from loc up to loc+args[0]*scale and down to loc+scale. Use loc=edges[0], scale=edges[-1] and args=[0.5] to define a symmetric triangle in the range of parameter P.</td>
</tr>
<tr>
<td>uniform</td>
<td>Flat from loc to loc+scale. Use loc=edges[0], scale=edges[-1] to define P as uniform over the range.</td>
</tr>
</tbody>
</table>

4.3. dist - Non-uniform samples
4.4 errors - Plot sample profile uncertainty

Visual representation of model uncertainty.

For reflectivity models, this aligns and plots a set of profiles chosen from the parameter uncertainty distribution, and plots the distribution of the residual values.

Use `run_errors` in a model file to reload the results of a batch DREAM fit.

```
refl1d.errors.reload_errors(model='', store='', nshown=50, random=True)
```

Reload the MCMC state and compute the model confidence intervals.

- `model` is the name of the model python file
- `store` is the name of the store directory containing the dream results
- `nshown` and `random` are as for `calc_errors_from_state()`.

Returns `errs` for `show_errors()`.

```
refl1d.errors.run_errors(**kw)
```

Argument parser for generating error plots from models.

The model directory should contain a fake model align.py with:

```
from refl1d.errors import run_errors
run_errors(model='', store='', align='auto')
```

If you are using the command line then you should be able to type the following at the command prompt to generate the plots:

```
$ refl1d align.py <model>.py <store> [align] [0112ln]
```

If you are using the GUI, you will have to set model, store and align directly in align.py each time you run.

Align is either auto for the current behaviour, or it is an interface number. You can align on the center of a layer by adding 0.5 to the interface number. You can count interfaces from the surface by prefixing with R. For example, 0 is the substrate interface, R1 is the surface interface, 2.5 is the the middle of layer 2 above the substrate.

You can plot the profiles and residuals on one plot by setting plots to 1, on two separate plots by setting plots to 2, or each curve on its own plot by setting plots to n. Plots are saved in `<store>/<model>-err#.png`. If plots is 0, then no plots are created.

Additional parameters include:

- `nshown`, `random`:...
see bumps.errplot.calc_errors_from_state()

contours, npoints, plots, save :
see show_errors()

refl1d.errors.calc_errors(problem, points)
Align the sample profiles and compute the residual difference from the measured reflectivity for a set of points.
The points should be sampled from the posterior probability distribution computed from MCMC, bootstrapping or sampled from the error ellipse calculated at the minimum.
Each of the returned arguments is a dictionary mapping model number to error sample data as follows:
Returns (profiles, slabs, Q, residuals).

profiles
Arrays of (z, rho, irho) for non-magnetic models or arrays of (z, rho, irho, rhoM, thetaM) for magnetic models. There will be one set of arrays returned per error sample.

slabs
Array of slab thickness for the layers in the models. There will be one array returned per error sample.
Using slab thickness, profiles can be aligned on interface boundaries and layer centers.

Q
Array of Q values for the data points in the model. The data points are the same for all error samples, so only one Q array is needed per model.

residuals
Array of (theory-data)/uncertainty for each data point in the measurement. There will be one array returned per error sample.

refl1d.errors.align_profiles(profiles, slabs, align)
Align profiles for each sample

refl1d.errors.show_errors(errors, contours=[68, 95], npoints=200, align='auto', plots=1, save=None)
Plot the aligned profiles and the distribution of the residuals for profiles and residuals returned from calc_errors.
contours can be a list of percentiles or []. If percentiles are given, then show uncertainty using a contour plot with the given levels, otherwise just overplot sample lines. contours defaults to [68, 95, 100].

npoints is the number of points to use when generating the profile contour. Since the z values for the various lines do not correspond, the contour generator interpolates the entire profile range with linear spacing using this number of points.

align is the interface number plus fractional distance within the layer following the interface. For example, use 0 for the substrate interface, use -1 for the surface interface, or use 2.5 for the center of the second slab above the substrate.

plots is the number of plots to use (1, 2, or ‘n’).

save is the basename of the plot to save. This should usually be “<store>/<model>”’. The program will add ‘-err#.png’ where ‘#' is the number of the plot.

refl1d.errors.show_profiles(errors, align, contours, npoints)
refl1d.errors.show_residuals(errors, contours)
4.5 experiment - Reflectivity fitness function

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Theory calculator.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExperimentBase</td>
<td>Support composite sample reflectivity measurements.</td>
</tr>
<tr>
<td>MixedExperiment</td>
<td>Fix v to a value with a given number of digits of precision</td>
</tr>
<tr>
<td>plot_sample</td>
<td>Quick plot of a reflectivity sample and the corresponding reflectivity.</td>
</tr>
</tbody>
</table>

Experiment definition

An experiment combines the sample definition with a measurement probe to create a fittable reflectometry model.

```python
class refl1d.experiment.Experiment(sample=None, probe=None, name=None, roughness_limit=0, dz=None, dA=None, step_interfaces=False, smoothness=None, interpolation=0)
```

Bases: refl1d.experiment.ExperimentBase

Theory calculator. Associates sample with data, Sample plus data. Associate sample with measurement.

The model calculator is specific to the particular measurement technique that was applied to the model.

Measurement properties:

- **probe** is the measuring probe

Sample properties:

- **sample** is the model sample
- **step_interfaces** use slabs to approximate gaussian interfaces
- **roughness_limit** limit the roughness based on layer thickness
- **dz** minimum step size for computed profile
- **smoothness** for computed profiles

If **step_interfaces** is True, then approximate the interface using microslabs with step size **dz**. The microslabs extend throughout the whole profile, both the interfaces and the bulk; a value for **dA** should be specified to save computation time. If False, then use the Nevot-Croce analytic expression for the interface between slabs.

The **roughness_limit** value should be reasonably large (e.g., 2.5 or above) to make sure that the Nevot-Croce reflectivity calculation matches the calculation of the displayed profile. Use a value of 0 if you want no limits on the roughness, but be aware that the displayed profile may not reflect the actual scattering densities in the material.

The **dz** step size sets the size of the slabs for non-uniform profiles. Using the relation \( d = 2 \pi / Q_{\text{max}} \), we use a default step size of \( d/20 \) rounded to two digits, with 5 Å as the maximum default. For simultaneous fitting you may want to set \( dz \) explicitly using to round(\( \pi/Q_{\text{max}}/10,1 \)) so that all models use the same step size.

The **dA** condition measures the uncertainty in scattering materials allowed when combining the steps of a non-uniform profile into slabs. Specifically, the area of the box containing the minimum and the maximum of the non-uniform profile within the slab will be smaller than **dA**. A **dA** of 10 gives coarse slabs. If **dA** is not provided then each profile step forms its own slab. The **dA** condition will also apply to the slab approximation to the interfaces.

**interpolation** indicates the number of points to plot in between existing points.

```python
amplitude (resolution=False)
```

- Calculate reflectivity amplitude at the probe points.

```python
format_parameters ()
```

- **is_reset()**

  Returns True if a model reset was triggered.
ismagnetic

magnetic_profile()
Return the nuclear and magnetic scattering potential for the sample.

magnetic_slabs()

name

nllf()
Return the -log(P(data|model)).

Using the assumption that data uncertainty is uncorrelated, with measurements normally distributed with
mean R and variance dR**2, this is just sum( resid**2/2 + log(2*pi*dR**2)/2 ).

The current version drops the constant term, sum(log(2*pi*dR**2)/2).

numpoints()

parameters()

penalty()

plot(plot_shift=None, profile_shift=None, view=None)

plot_profile(plot_shift=None)

plot_reflectivity(show_resolution=False, view=None, plot_shift=None)

profile_shift = 0

reflectivity(resolution=True, interpolation=0)
Calculate predicted reflectivity.

If resolution is true include resolution effects.

residuals()

restore_data()
Restore original data after resynthesis.

resynth_data()
Resynthesize data with noise from the uncertainty estimates.

save(basename)

save_profile(basename)

save_refl(basename)

save_staj(basename)

simulate_data(noise=2)
Simulate a random data set for the model

Parameters:

noise = 2 [float | %] Percentage noise to add to the data.

slabs()
Return the slab thickness, roughness, rho, irho for the rendered model.

Note: Roughness is for the top of the layer.
smooth_profile\( (dz=0.1) \)

Return the scattering potential for the sample.

If \(dz\) is not given, use \(dz = 0.1\) A.

step_profile()

Return the step scattering potential for the sample, ignoring interfaces.

update()

Called when any parameter in the model is changed.

This signals that the entire model needs to be recalculated.

update_composition()

When the model composition has changed, we need to lookup the scattering factors for the new model.

This is only needed when an existing chemical formula is modified; new and deleted formulas will be handled automatically.

write_data(filename, **kw)

Save simulated data to a file

class refl1d.experiment.ExperimentBase

Bases: object

format_parameters()

is_reset()

Returns True if a model reset was triggered.

name

nllf()

Return the \(-\log(P(\text{data|model}))\).

Using the assumption that data uncertainty is uncorrelated, with measurements normally distributed with mean R and variance \(dR^2\), this is just \(\text{sum}(\text{resid}^2/2 + \log(2\pi dR^2)/2)\).

The current version drops the constant term, \(\text{sum}(\log(2\pi dR^2)/2)\).

numpoints()

plot(plot_shift=None, profile_shift=None, view=None)

plot_reflectivity(show_resolution=False, view=None, plot_shift=None)

reflectivity(resolution=True, interpolation=0)

residuals()

restore_data()

Restore original data after resynthesis.

resynth_data()

Resynthesize data with noise from the uncertainty estimates.

save(basename)

save_profile(basename)

save_refl(basename)

simulate_data(noise=2)

Simulate a random data set for the model

Parameters:

\(noise = 2\) [float | \%] Percentage noise to add to the data.
**update()**
Called when any parameter in the model is changed.
This signals that the entire model needs to be recalculated.

**update_composition()**
When the model composition has changed, we need to lookup the scattering factors for the new model.
This is only needed when an existing chemical formula is modified; new and deleted formulas will be handled automatically.

**write_data(filename, **kw)**
Save simulated data to a file

```python
class refl1d.experiment.MixedExperiment(samples=None, ratio=None, probe=None, name=None, coherent=False, interpolation=0, **kw)
```
Bases: `refl1d.experiment.ExperimentBase`
Support composite sample reflectivity measurements.
Sometimes the sample you are measuring is not uniform. For example, you may have one portion of your polymer brush sample where the brushes are close packed and able to stay upright, whereas a different section of the sample has the brushes lying flat. Constructing two sample models, one with brushes upright and one with brushes flat, and adding the reflectivity incoherently, you can then fit the ratio of upright to flat.

- **samples** the layer stacks making up the models
- **ratio** a list of parameters, such as [3,1] for a 3:1 ratio
- **probe** the measurement to be fitted or simulated
- **coherent** is True if the length scale of the domains is less than the coherence length of the neutron, or false otherwise.

Statistics such as the cost functions for the individual profiles can be accessed from the underlying experiments using composite.parts[i] for the various samples.

**amplitude(resolution=False)**

**format_parameters()**

**is_reset()**
Returns True if a model reset was triggered.

**name**

**nllf()**
Return the \(-\log(P(\text{data|model}))\).
Using the assumption that data uncertainty is uncorrelated, with measurements normally distributed with mean R and variance dR**2, this is just sum( resid**2/2 + log(2*pi*dR**2)/2 ).
The current version drops the constant term, sum(log(2*pi*dR**2)/2).

**numpoints()**

**parameters()**

**penalty()**

**plot(plot_shift=None, profile_shift=None, view=None)**

**plot_profile(plot_shift=None)**

**plot_reflectivity(show_resolution=False, view=None, plot_shift=None)**

**reflectivity(resolution=True, interpolation=0)**
Calculate predicted reflectivity.
This will be the weighted sum of the reflectivity from the individual systems. If coherent is set, then the coherent sum will be used, otherwise the incoherent sum will be used.

If resolution is true include resolution effects.

interpolation is the number of theory points to show between data points.

residuals()

restore_data()

Reserve original data after resynthesis.

resynth_data()

Resynthesize data with noise from the uncertainty estimates.

save (basename)

save_profile (basename)

save_refl (basename)

save_staj (basename)

simulate_data (noise=2)

Simulate a random data set for the model

Parameters:

noise = 2  [float | %] Percentage noise to add to the data.

update()

update_composition()

When the model composition has changed, we need to lookup the scattering factors for the new model.
This is only needed when an existing chemical formula is modified; new and deleted formulas will be handled automatically.

write_data (filename, **kw)

Save simulated data to a file

refl1d.experiment.nice (v, digits=2)

Fix v to a value with a given number of digits of precision

refl1d.experiment.plot_sample (sample, instrument=None, roughness_limit=0)

Quick plot of a reflectivity sample and the corresponding reflectivity.

4.6 fitplugin - Bumps plugin definition for reflectivity models

<table>
<thead>
<tr>
<th>data_view</th>
<th>model_view</th>
<th>new_model</th>
</tr>
</thead>
<tbody>
<tr>
<td>calc_errors</td>
<td>show_errors</td>
<td>Align the sample profiles and compute the residual difference from the measured reflectivity for a set of points.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Plot the aligned profiles and the distribution of the residuals for profiles and residuals returned from calc_errors.</td>
</tr>
</tbody>
</table>

Reflectivity plugin for fitting GUI.

Note that the fitting infrastructure is still heavily tied to the reflectivity modeling program, and this represents only the first tiny steps to separating the two.

refl1d.fitplugin.data_view()
Align the sample profiles and compute the residual difference from the measured reflectivity for a set of points.

The points should be sampled from the posterior probability distribution computed from MCMC, bootstrapping or sampled from the error ellipse calculated at the minimum.

Each of the returned arguments is a dictionary mapping model number to error sample data as follows:

Returns (profiles, slabs, Q, residuals).

- **profiles**
  Arrays of \((z, \rho, \rho^*)\) for non-magnetic models or arrays of \((z, \rho, \rho^*, \rho_M, \theta_M)\) for magnetic models. There will be one set of arrays returned per error sample.

- **slabs**
  Array of slab thickness for the layers in the models. There will be one array returned per error sample. Using slab thickness, profiles can be aligned on interface boundaries and layer centers.

- **Q**
  Array of Q values for the data points in the model. The data points are the same for all error samples, so only one Q array is needed per model.

- **residuals**
  Array of \((\text{theory-data})/\text{uncertainty}\) for each data point in the measurement. There will be one array returned per error sample.

Plot the aligned profiles and the distribution of the residuals for profiles and residuals returned from \texttt{calc_errors}.

- **contours** can be a list of percentiles or \([\,]\). If percentiles are given, then show uncertainty using a contour plot with the given levels, otherwise just overplot sample lines. \texttt{contours} defaults to \([68, 95, 100]\).

- **npoints** is the number of points to use when generating the profile contour. Since the \(z\) values for the various lines do not correspond, the contour generator interpolates the entire profile range with linear spacing using this number of points.

- **align** is the interface number plus fractional distance within the layer following the interface. For example, use 0 for the substrate interface, use -1 for the surface interface, or use 2.5 for the center of the second slab above the substrate.

- **plots** is the number of plots to use (1, 2, or ‘n’).

- **save** is the basename of the plot to save. This should usually be `<store>/<model>`. The program will add ‘-err#.png’ where ‘#’ is the number of the plot.

### 4.7 freeform - Freeform - Parametric B-Spline

| FreeInterface | A freeform section of the sample modeled with monotonic splines. |
| FreeLayer | A freeform section of the sample modeled with B-splines. |
| FreeformInterface01 | A freeform section of the sample modeled with B-splines. |

Freeform modeling with B-Splines
class reflid.freeform.FreeInterface(interface=0, below=None, above=None, dz=None, dp=None, name='Interface')

Bases: reflid.model.Layer

A freeform section of the sample modeled with monotonic splines.

Layers have a slope of zero at the ends, so they automatically blend with slabs.

class reflid.freeform.FreeLayer(thickness=0, left=None, right=None, rho=[], irho=[], rhoz=[], irhoz=[], name='Freeform')

Bases: reflid.model.Layer

A freeform section of the sample modeled with B-splines.

sld (rho) and imaginary sld (irho) can be modeled with a separate number of control points. The control points can be equally spaced in the layers unless rhoz or irhoz are specified. If the z values are given, they must be in the range [0,1]. One control point is anchored at either end, so there are two fewer z values than controls if z values are given.

Layers have a slope of zero at the ends, so they automatically blend with slabs.
parameters()

penalty()

Return a penalty value associated with the layer. This should be zero if the parameters are valid, and increasing as the parameters become more invalid. For example, if total volume fraction exceeds unity, then the penalty would be the amount by which it exceeds unity, or if z values must be sorted, then penalty would be the amount by which they are unsorted.

Note that penalties are handled separately from any probability of seeing a combination of layer parameters; the final solution to the problem should not include any penalized points.

render(probe, slabs)

thickness = None

class reflid.freeform.FreeformInterface01(thickness=0, interface=0, below=None, above=None, z=None, vf=None, name='Interface')

Bases: reflid.model.Layer

A freeform section of the sample modeled with B-splines.

sld (rho) and imaginary sld (irho) can be modeled with a separate number of control points. The control points can be equally spaced in the layers unless rhoz or irhoz are specified. If the z values are given, they must be in the range [0,1]. One control point is anchored at either end, so there are two fewer z values than controls if z values are given.

Layers have a slope of zero at the ends, so the automatically blend with slabs.

cd

constraints()

Constraints

find(z)

Find the layer at depth z.

Returns layer, start, end

interface = None

ismagnetic

layer_parameters()

magnetism

parameters()

penalty()

Return a penalty value associated with the layer. This should be zero if the parameters are valid, and increasing as the parameters become more invalid. For example, if total volume fraction exceeds unity, then the penalty would be the amount by which it exceeds unity, or if z values must be sorted, then penalty would be the amount by which they are unsorted.

Note that penalties are handled separately from any probability of seeing a combination of layer parameters; the final solution to the problem should not include any penalized points.

render(probe, slabs)

thickness = None

4.8 fresnel - Pure python Fresnel reflectivity calculator
Pure python Fresnel reflectivity calculator.

class refl1d.fresnel.Fresnel (rho=0, irho=0, sigma=0, Vrho=0, Virho=0)

Function for computing the Fresnel reflectivity for a single interface.

Parameters

rho, irho = 0 [float | 1e6 * inv Angstrom^2] real and imaginary scattering length density of backing medium

Vrho, Virho = 0 [float | 1e6 * inv Angstrom^2] real and imaginary scattering length density of incident medium

sigma = 0 [float | Angstrom] interfacial roughness

Returns

fresnel [Fresnel] callable object for computing Fresnel reflectivity at Q

Note that we do not correct for attenuation of the beam through the incident medium since we do not know the path length.

reflectivity (Q)

Compute the Fresnel reflectivity at the given Q/wavelength.

refl1d.fresnel.test()

4.9 garefl - Adaptor for garefl models

Load garefl models into refl1d.

The models themselves don’t need to be modified. See the garefl documentation for setting up the model.

One extension provided to refl1d that is not available in garefl is the use of penalty values in the constraints. The model constraints is able to set:

fit[0].penalty = FIT_REJECT_PENALTY + distance

Here, distance is the distance to the valid region of the search space so that any fitter that gets lost in a penalty region can more quickly return to the valid region. Any penalty value above FIT_REJECT_PENALTY will suppress the evaluation of the model at that point during the fit.

Consider a model with layers (Si | Au | FeNi | air) and the constraint that d_Au + d_FeNi < 200 A. The constraints function would be written something like:

double excess = fit[0].m.d[1] + fit[0].m.d[2] - 200;
fit[0].penalty = excess > 0 ? excess*excess+FIT_REJECT_PENALTY : 0.;

Then, if the fit algorithm proposes a value such as Au=125, FeNi=90, the excess will be 15, and the penalty will be FIT_REJECT_PENALTY+225.

You can use penalties less than FIT_REJECT_PENALTY, but these should correspond to the negative log likelihood of seeing that constraint value within the model in order for the MCMC uncertainty analysis to work correctly.
Ref1D: Neutron and X-Ray Reflectivity Analysis, Release 0.7.8

FIT_REJECT_PENALTY is set to 1e6, which should be high enough that it doesn’t perturb the fit.

test.garefl.load(modelfile)

4.10 instrument - Reflectivity instrument definition

<table>
<thead>
<tr>
<th>Monochromatic</th>
<th>Instrument representation for scanning reflectometers.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pulsed</td>
<td>Instrument representation for pulsed reflectometers.</td>
</tr>
</tbody>
</table>

Reflectometry instrument definitions.

An instrument definition contains all the information necessary to compute the resolution for a measurement. See resolution for details.

This module is intended to help define new instrument loaders.

4.10.1 Scanning Reflectometers

ref1d.instrument (this module) defines two instrument types: Monochromatic and Pulsed. These represent generic scanning and time of flight instruments, respectively.

To perform a simulation or load a data set, a measurement geometry must be defined. In the following example, we set up the geometry for a pretend instrument SP:2. The complete geometry needs to include information to calculate wavelength resolution (wavelength and wavelength dispersion) as well as angular resolution (slit distances and openings, and perhaps sample size and sample warp). In this case, we are using a scanning monochromatic instrument with slits of 0.1 mm below 0.5° and opening slits above 0.5° starting at 0.2 mm. The monochromatic instrument assumes a fixed ∆θ/θ while opening.

```python
from refl1d.names import *
geometry = Monochromatic(instrument="SP:2", radiation="neutron",
... wavelength=5.0042, dLoL=0.009, d_s1=230+1856, d_s2=230,
... Tlo=0.5, slits_at_Tlo=0.2, slits_below=0.1)
```

This instrument can be used to a data file, or generate a measurement probe for use in modeling or to read in a previously measured data set or generate a probe for simulation:

```python
from numpy import linspace, loadtxt

datafile = sample_data('10ndt001.refl')

Q,R,dR = loadtxt(datafile).T

probe = geometry.probe(Q=Q, data=(R,dR))
simulation = geometry.probe(T=linspace(0,5,51))
```

All instrument parameters can be specified when constructing the probe, replacing the defaults that are associated with the instrument. For example, to include sample broadening effects in the resolution:

```python
probe2 = geometry.probe(Q=Q, data=(R,dR), sample_broadening=0.1,
... name="probe2")
```

For magnetic systems a polarized beam probe is needed:

```python
magnetic_probe = geometry.magnetic_probe(T=numpy.linspace(0,5,100),)
```

The string representation of the geometry prints a multi-line description of the default instrument configuration:
4.10.2 Predefined Instruments

Specific instruments can be defined for each facility. This saves the users having to remember details of the instrument geometry.

For example, the above SP:2 instrument could be defined as follows:

```python
class SP2(Monochromatic):
    ...
    instrument = "SP:2"
    ...
    radiation = "neutron"
    ...
    wavelength = 5.0042  # Angstroms
    ...
    dLoL = 0.009  # FWHM
    ...
    d_s1 = 230.0 + 1856.0  # mm
    ...
    d_s2 = 230.0  # mm
    ...
    def load(self, filename, **kw):
        ...
        Q,R,dR = loadtxt(datafile).T
        ...
        probe = self.probe(Q=Q, data=(R,dR), **kw)
        ...
        return probe
```

This definition can then be used to define the measurement geometry. We have added a load method which knows about the facility file format (in this case, three column ASCII data Q, R, dR) so that we can load a datafile in a couple of lines of code:

```python
>>> geometry = SP2(Tlo=0.5, slits_at_Tlo=0.2, slits_below=0.1)
>>> probe3 = geometry.load(datafile)
```

The defaults() method prints the static components of the geometry:

```python
>>> print(SP2defaults())
== Instrument class SP:2 ==
radiation = neutron at 5.0042 Angstrom with 0.9% resolution
slit distances = 2086 mm and 230 mm
```

4.10.3 GUI Usage

Graphical user interfaces follow different usage patterns from scripts. Here the emphasis will be on selecting a data set to process, displaying its default metadata and allowing the user to override it.

File loading should follow the pattern established in reflectometry reduction, with an extension registry and a fallback scheme whereby files can be checked in a predefined order. If the file cannot be loaded, then the next loader is tried. This should be extended with the concept of a magic signature such as those used by graphics and sound file applications: read the first block and run it through the signature check before trying to load it. For unrecognized extensions, all loaders can be tried.

The file loader should return an instrument instance with metadata initialized from the file header. This metadata can be displayed to the user along with a plot of the data and the resolution. When metadata values are changed, the
resolution can be recomputed and the display updated. When the data set is accepted, the final resolution calculation can be performed.

```python
class refl1d.instrument.Monochromatic(**kw)
    Bases: object

    Instrument representation for scanning reflectometers.

    Parameters

    instrument [string] name of the instrument
    radiation [string | xray or neutron] source radiation type
    d_s1, d_s2 [float | mm] distance from sample to pre-sample slits 1 and 2; post-sample slits are ignored
    wavelength [float | Å] wavelength of the instrument
    dLoL [float] constant relative wavelength dispersion; wavelength range and dispersion together determine the bins
    slits [float OR (float,float) | mm] fixed slits
    slits_at_Tlo [float OR (float,float) | mm] slit 1 and slit 2 openings at Tlo; this can be a scalar if both slits are open by the same amount, otherwise it is a pair (s1,s2).
    slits_at_Qlo [float OR (float,float) | mm] equivalent to slits_at_Tlo, for instruments that are controlled by Q rather than theta
    Tlo, Thi [float | ∘] range of opening slits, or inf if slits are fixed.
    Qlo, Qhi [float | Å⁻¹] range of opening slits when instrument is controlled by Q.
    slits_below, slits_above [float OR (float,float) | mm] slit 1 and slit 2 openings below Tlo and above Thi; again, these can be scalar if slit 1 and slit 2 are the same, otherwise they are each a pair (s1,s2). Below and above default to the values of the slits at Tlo and Thi respectively.
    sample_width [float | mm] width of sample; at low angle with tiny samples, stray neutrons miss the sample and are not reflected onto the detector, so the sample itself acts as a slit, therefore the width of the sample may be needed to compute the resolution correctly
    sample_broadening [float | ∘ FWHM] amount of angular divergence (+) or focusing (-) introduced by the sample; this is caused by sample warp, and may be read off of the rocking curve by subtracting (s1+s2)/2/(d_s1-d_s2) from the FWHM width of the rocking curve

    Thi = 90
    Tlo = 90

calc_dt(**kw)

    Compute the angular divergence for given slits and angles

    Parameters

    T OR Q [float | ∘ OR Å⁻¹] measurement angles
    slits [float OR (float,float) | mm] total slit opening from edge to edge, not beam center to edge
    d_s1, d_s2 [float | mm] distance from sample to slit 1 and slit 2
    sample_width [float | mm] size of sample
    sample_broadening [float | ∘ FWHM] resolution changes from sample warp

    Returns
```

4.10. instrument - Reflectivity instrument definition
sample_broadening can be estimated from W, the full width at half maximum of a rocking curve measured in degrees:

\[
\text{sample_broadening} = W - \text{degrees} \left( \frac{0.5*(s1+s2)}{d1-d2} \right)
\]

calc_slits (**kw)

Determines slit openings from measurement pattern.

If slits are fixed simply return the same slits for every angle, otherwise use an opening range \([Tlo, Thi]\) and the value of the slits at the start of the opening to define the slits. Slits below Tlo and above Thi can be specified separately.

\(T\) OR \(Q\) incident angle or \(Q\ Tlo, Thi\) angle range over which slits are opening \(slits\_at\_Tlo\) openings at the start of the range, or fixed opening \(slits\_below, slits\_above\) openings below and above the range

Use fixed_slits is available, otherwise use opening slits.

dLoL = None
d_s1 = None
d_s2 = None

classmethod defaults ()

Return default instrument properties as a printable string.

fixed_slits = None

instrument = ‘monochromatic’

magnetic_probe(Aguide=270, shared_beam=True, H=0, **kw)

Simulate a polarized measurement probe.

Returns a probe with \(Q\), angle, wavelength and the associated uncertainties, but not any data.

Guide field angle \(Aguide\) can be specified, as well as keyword arguments for the geometry of the probe cross sections such as \(slits\_at\_Tlo, Tlo, Thi, slits\_below, and slits\_above\) to define the angular divergence.

probe (**kw)

Return a probe for use in simulation.

Parameters

\(Q\) \([\text{float}] \text{ Å}\) Q values to be measured.

\(T\) \([\text{float}] \text{ °}\) Angles to be measured.

Additional keyword parameters

Returns

probe \([\text{Probe}]\) Measurement probe with complete resolution information. The probe will not have any data.

If both \(Q\) and \(T\) are specified then \(Q\) takes precedents.

You can override instrument parameters using key=value. In particular, settings for \(slits\_at\_Tlo, Tlo, Thi, slits\_below, \) and \(slits\_above\) are used to define the angular divergence.

radiation = ‘unknown’

resolution (**kw)

Calculate resolution at each angle.

Return
$$T, dT \quad [\text{float} | ^\circ] \text{ Angles and angular divergence.}$$

$$L, dL \quad [\text{float} | \text{Å}] \text{ Wavelengths and wavelength dispersion.}$$

```python
sample_broadening = 0
sample_width = 10000000000.0
slits_above = None
slits_at_Tlo = None
slits_below = None
wavelength = None
```

```python
class refl1d.instrument.Pulsed(**kw)
Bases: object

Instrument representation for pulsed reflectometers.

Parameters

- **instrument** [string] name of the instrument
- **radiation** [string | xray, neutron] source radiation type
- **TOF_range** [(float, float)] usable range of times for TOF data
- **T** [float | ^\circ] sample angle
- **d_s1, d_s2** [float | mm] distance from sample to pre-sample slits 1 and 2; post-sample slits are ignored
- **wavelength** [(float, float) | Å] wavelength range for the measurement
- **dLoL** [float] constant relative wavelength dispersion; wavelength range and dispersion together determine the bins
- **slits** [float OR (float, float) | mm] fixed slits
- **slits_at_Tlo** [float OR (float, float) | mm] slit 1 and slit 2 openings at Tlo; this can be a scalar if both slits are open by the same amount, otherwise it is a pair (s1,s2).
- **Tlo, Thi** [float | ^\circ] range of opening slits, or inf if slits are fixed.
- **slits_below, slits_above** [float OR (float, float) | mm] slit 1 and slit 2 openings below Tlo and above Thi; again, these can be scalar if slit 1 and slit 2 are the same, otherwise they are each a pair (s1,s2). Below and above default to the values of the slits at Tlo and Thi respectively.
- **sample_width** [float | mm] width of sample; at low angle with tiny samples, stray neutrons miss the sample and are not reflected onto the detector, so the sample itself acts as a slit, therefore the width of the sample may be needed to compute the resolution correctly
- **sample_broadening** [float | ^\circ FWHM] amount of angular divergence (+) or focusing (-) introduced by the sample; this is caused by sample warp, and may be read off of the rocking curve by subtracting 0.5*(s1+s2)/(d_s1-d_s2) from the FWHM width of the rocking curve

```

```python
T = None
TOF_range = (0, inf)
Thi = 90
Tlo = 90
calc_dT(T, slits, **kw)
```

4.10. instrument - Reflectivity instrument definition
calc_slits(**kw)
Determines slit openings from measurement pattern.

If slits are fixed simply return the same slits for every angle, otherwise use an opening range [Tlo, Thi] and the value of the slits at the start of the opening to define the slits. Slits below Tlo and above Thi can be specified separately.

T incident angle Tlo, Thi angle range over which slits are opening slits_at_Tlo openings at the start of the range, or fixed opening slits_below, slits_above openings below and above the range

Use fixed_slits is available, otherwise use opening slits.

dLoL = None
d_s1 = None
d_s2 = None

classmethod defaults()  
Return default instrument properties as a printable string.

fixed_slits = None

instrument = ‘pulsed’

magnetic_probe(Aguide=270, shared_beam=True, **kw)
Simulate a polarized measurement probe.

Returns a probe with Q, angle, wavelength and the associated uncertainties, but not any data.

Guide field angle Aguide can be specified, as well as keyword arguments for the geometry of the probe cross sections such as slit settings slits and T to define the angular divergence and dLoL to define the wavelength resolution.

probe(**kw)
Simulate a measurement probe.

Returns a probe with Q, angle, wavelength and the associated uncertainties, but not any data.

You can override instrument parameters using key=value. In particular, slit settings slits and T define the angular divergence and dLoL defines the wavelength resolution.

radiation = ‘neutron’

resolution(L, dL, **kw)
Return the resolution of the measurement. Needs T, L, dL specified as keywords.

sample_broadening = 0

sample_width = 10000000000.0

simulate(sample, uncertainty=1, **kw)
Simulate a run with a particular sample.

Parameters

sample  [Stack] Reflectometry model

T   [[float] | °] List of angles to be measured, such as [0.15,0.4,1,2].

slits  [[float] or [(float,float)] | mm] Slit settings for each angle.

uncertainty = 1  [float or [float] | %] Incident intensity is set so that the median dR/R is equal to uncertainty, where R is the idealized reflectivity of the sample.

dLoL = 0.02: float  Wavelength resolution
normalize = True  [boolean] Whether to normalize the intensities

theta_offset = 0  [float] Sample alignment error

background = 0  [float] Background counts per incident neutron (background is assumed to be independent of measurement geometry).

back_reflectivity = False  [boolean] Whether beam travels through incident medium or through substrate.


slits = None
slits_above = None
slits_at_Tlo = None
slits_below = None
wavelength = None

4.11 magnetism - Magnetic Models

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseMagnetism</td>
<td>Magnetic properties of the layer.</td>
</tr>
<tr>
<td>FreeMagnetism</td>
<td>Spline change in magnetism throughout layer.</td>
</tr>
<tr>
<td>Magnetism</td>
<td>Region of constant magnetism.</td>
</tr>
<tr>
<td>MagnetismStack</td>
<td>Magnetic slabs within a magnetic layer.</td>
</tr>
<tr>
<td>MagnetismTwist</td>
<td>Linear change in magnetism throughout layer.</td>
</tr>
</tbody>
</table>

Magnetic modeling for 1-D reflectometry.

Magnetic properties are tied to the structural description of the but only loosely.

There may be dead regions near the interfaces of magnetic materials.

Magnetic behaviour may be varying in complex ways within and across structural boundaries. For example, the ma Indeed, the pattern may continue across spacer layers, going to zero in the magnetically dead region and returning to its long range variation on entry to the next magnetic layer. Magnetic multilayers may exhibit complex magnetism throughout the repeated section while the structural components are fixed.

The scattering behaviour is dependent upon net field strength relative to polarization direction. This arises from three underlying quantities: the strength of the individual dipole moments in the layer, the degree of alignment of these moments, and the net direction of the alignment. The strength of the dipole moment depends on the details of the electronic structure, so unlike the nuclear scattering potential, it cannot be readily determined from material composition. Similarly, net magnetization depends on the details of the magnetic domains within the material, and cannot readily be determined from first principles. The interaction potential of the net magnetic moment depends on the alignment of the field with respect to the beam, with a net scattering length density of :math:`\rho_0 M \cos(\theta_M)`. Clearly the scattering measurement will not be able to distinguish between a reduced net magnetic strength :math:`\rho_0 M` and a change in orientation :math:`\theta_M` for an individual measurement, as should be apparent from the correlated uncertainty plot produced when both parameters are fit.

Magnetism support is split into two parts: describing the layers and anchoring them to the structure.

```python
class refl1d.magnetism.BaseMagnetism:
    def __init__(self, extent=1, dead_below=0, dead_above=0, interface_below=None, interface_above=None, name='LAYER'):
        Bases: object
```

4.11. magnetism - Magnetic Models
Magnetic properties of the layer.

Magnetism is attached to set of nuclear layers by setting the magnetism property of the first layer to the rendered for the magnetic profile, and setting the magnetism.extent property to say how many layers it extends over.

dead_below and dead_above are dead regions within the magnetic extent, which allow you to shift the magnetic interfaces relative to the nuclear interfaces.

interface_below and interface_above are the interface widths for the magnetic layer, which default to the inter-face widths for the corresponding nuclear layers if no interfaces are specified. For consecutive layers, only interface_above is used; any value for interface_below is ignored.

parameters()

set_layer_name(name)

Update the names of the magnetic parameters with the name of the layer if it has not already been set. This is necessary since we don’t know the layer name until after we have constructed the magnetism object.

class refl1d.magnetism.FreeMagnetism(z=[], rhoM=[], thetaM=[], name='LAYER', **kw)

Bases: refl1d.magnetism.BaseMagnetism

Spline change in magnetism throughout layer.

magnetic = True

parameters()

profile(Pz, thickness)

render (probe, slabs, thickness, anchor, sigma)

set_layer_name(name)

Update the names of the magnetic parameters with the name of the layer if it has not already been set. This is necessary since we don’t know the layer name until after we have constructed the magnetism object.

class refl1d.magnetism.Magnetism(rhoM=0, thetaM=270, name='LAYER', **kw)

Bases: refl1d.magnetism.BaseMagnetism

Region of constant magnetism.

parameters()

render (probe, slabs, thickness, anchor, sigma)

set_layer_name(name)

Update the names of the magnetic parameters with the name of the layer if it has not already been set. This is necessary since we don’t know the layer name until after we have constructed the magnetism object.

class refl1d.magnetism.MagnetismStack(weight=[], rhoM=[], thetaM=[270], interfaceM=[0], name='LAYER', **kw)

Bases: refl1d.magnetism.BaseMagnetism

Magnetic slabs within a magnetic layer.

parameters()

render (probe, slabs, thickness, anchor, sigma)

set_layer_name(name)

Update the names of the magnetic parameters with the name of the layer if it has not already been set. This is necessary since we don’t know the layer name until after we have constructed the magnetism object.

class refl1d.magnetism.MagnetismTwist(rhoM=[0, 0], thetaM=[270, 270], name='LAYER', **kw)

Bases: refl1d.magnetism.BaseMagnetism
Linear change in magnetism throughout layer.

```python
magnetic = True
parameters()
render (probe, slabs, thickness, anchor, sigma)
set_layer_name (name)
```

Update the names of the magnetic parameters with the name of the layer if it has not already been set. This is necessary since we don’t know the layer name until after we have constructed the magnetism object.

### 4.12 material - Material

<table>
<thead>
<tr>
<th>Material</th>
<th>Description of a solid block of material.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mixture</td>
<td>Mixed block of material.</td>
</tr>
<tr>
<td>SLD</td>
<td>Unknown composition.</td>
</tr>
<tr>
<td>Vacuum</td>
<td>Empty layer</td>
</tr>
<tr>
<td>Scatterer</td>
<td>A generic scatterer separates the lookup of the scattering factors from the calculation of the scattering length density.</td>
</tr>
<tr>
<td>ProbeCache</td>
<td>Probe proxy for materials properties.</td>
</tr>
</tbody>
</table>

Reflectometry materials.

Materials (see **Material**) have a composition and a density. Density may not be known, either because it has not been measured or because the measurement of the bulk value does not apply to thin films. The density parameter can be fitted directly, or the bulk density can be used, and a stretch parameter can be fitted.

Mixtures (see **Mixture**) are a special kind of material which are composed of individual parts in proportion. A mixture can be constructed in a number of ways, such as by measuring proportional masses and mixing or measuring proportional volumes and mixing. The parameter of interest may also be the relative number of atoms of one material versus another. The fractions of the different mixture components are fitted parameters, with the remainder of the bulk filled by the final component.

SLDs (see **SLD**) are raw scattering length density values. These should be used if the material composition is not known. In that case, you will need separate SLD materials for each wavelength and probe.

`air` (see **Vacuum**) is a predefined scatterer transparent to all probes.

Scatter (see **Scatterer**) is the abstract base class from which all scatterers are derived.

The probe cache (see **ProbeCache**) stores the scattering factors for the various materials and calls the material sld method on demand. Because the same material can be used for multiple measurements, the scattering factors cannot be stored with material itself, nor does it make sense to store them with the probe. The scattering factor lookup for the material is separate from the scattering length density calculation so that you only need to look up the material once per fit.

The probe itself deals with all computations relating to the radiation type and energy. Unlike the normally tabulated scattering factors \( f', f'' \) for X-ray, there is no need to scale by probe by electron radius. In the end, sld is just the returned scattering factors times density.

```python
class refl1d.material.Material (formula=None, name=None, use_incoherent=False, density=None, natural_density=None, fitby='bulk_density', value=None)
```

Bases: refl1d.material.Scatterer

Description of a solid block of material.

**Parameters**

- `formula`: Formula
Composition can be initialized from either a string or a chemical formula. Valid values are defined in periodictable.formula.

\[ \text{density} : \text{float} \ | \ g \cdot \text{cm}^{-3} \]

If specified, set the bulk density for the material.

\[ \text{natural_density} : \text{float} \ | \ g \cdot \text{cm}^{-3} \]

If specified, set the natural bulk density for the material.

\[ \text{use_incoherent} = \text{False} : \text{boolean} \]

True if incoherent scattering should be interpreted as absorption.

\[ \text{fitby} = \text{‘bulk_density’} : \text{string} \]

Which density parameter is the fitting parameter. The choices are bulk_density, natural_density, relative_density or cell_volume. See fitby() for details.

\[ \text{value} : \text{Parameter} \text{ or float} \ | \ \text{units depends on fitby type} \]

Initial value for the fitted density parameter. If None, the value will be initialized from the material density.

For example, to fit Pd by cell volume use:

```python
>>> m = Material('Pd', fitby='cell_volume')
>>> m.cell_volume.range(1,10)
Parameter(Pd cell_volume)
>>> print("%.2f %.2f"%(m.density.value, m.cell_volume.value))
12.02 14.70
```

You can change density representation by calling material.fitby(type).

```python
fitby(type=’bulk_density’, value=None)
```

Specify the fitting parameter to use for material density.

**Parameters**

- **type** [string] Density representation
- **value** [Parameter] Initial value, or associated parameter.

Density type can be one of the following:

- **bulk_density** [g·cm\(^{-3}\) or kg/L] Density is bulk_density
- **natural_density** [g·cm\(^{-3}\) or kg/L] Density is natural_density / (natural mass/isotope mass)
- **relative_density** [unitless] Density is relative_density * formula density
- **cell_volume** [Å\(^3\)] Density is mass / cell_volume

The resulting material will have a density attribute with the computed material density in addition to the fitby attribute specified.

**Note:** Calling fitby replaces the density parameter in the material, so be sure to do so before using density in a parameter expression. Using bumps.parameter.WrappedParameter for density is another alternative.
class refl1d.material.Mixture(base, parts, by='volume', name=None, use_incoherent=False)

Bases: refl1d.material.Scatterer

Mixed block of material.

The components of the mixture can vary relative to each other, either by mass, by volume or by number:

Mixture.bymass(base, M1, F1, M2, F2..., name='mixture name')
Mixture.byvolume(base, M1, F1, M2, F2..., name='mixture name')

The materials base, M1, M2, M3, ... can be chemical formula strings or material objects. In practice, since the chemical formula parser does not have a density database, only elemental materials can be specified by string. Use natural_density will need to change from bulk values if the formula has isotope substitutions.

The fractions F2, F3, ... are percentages in [0,100]. The implicit fraction F1 is 100 - (F2+F3+...). The SLD is NaN when F1 < 0).

name defaults to M1.name+M2.name+...

classmethod bymass(base, *parts, **kw)
    Returns an alloy defined by relative mass of the constituents.
    Mixture.bymass(base, M1, F2,..., name='mixture name')

classmethod byvolume(base, *parts, **kw)
    Returns an alloy defined by relative volume of the constituents.
    Mixture.byvolume(M1, M2, F2,..., name='mixture name')

density
    Compute the density of the mixture from the density and proportion of the individual components.

parameters()
    Adjustable parameters are the fractions associated with each constituent and the relative scale fraction used to tweak the overall density.

sld(probe)
    Return the scattering length density and absorption of the mixture.

class refl1d.material.SLD(name='SLD', rho=0, irho=0)
Bases: refl1d.material.Scatterer

Unknown composition.

Use this when you don’t know the composition of the sample. The absorption and scattering length density are stored directly rather than trying to guess at the composition from details about the sample.

The complex scattering potential is defined by \( \rho + j \rho_i \). Note that this differs from \( \rho + j \mu / (2 \lambda) \) more traditionally used in neutron reflectometry, and \( N_r (f_1 + j f_2) \) traditionally used in X-ray reflectometry.

Given that \( f_1 \) and \( f_2 \) are always wavelength dependent for X-ray reflectometry, it will not make much sense to use this for wavelength varying X-ray measurements. Similarly, some isotopes, particularly rare earths, show wavelength dependence for neutrons, and so time-of-flight measurements should not be fit with a fixed SLD scatterer.

parameters()

sld(probe)

class refl1d.material.Vacuum
Bases: refl1d.material.Scatterer

Empty layer

name = ‘air’
**sld**

Return the scattering length density expected for the given scattering factors, as returned from a call to scattering_factors() for a particular probe.

### `refl1d.material.ProbeCache` *(probe=None)*

Probes proxy for materials properties.

A caching probe which only looks up scattering factors for materials which it hasn’t seen before. Note that caching is based on object id, and will fail if the material object is updated with a new atomic structure.

`probe` is the probe to use when looking up the scattering length density.  

The scattering factors need to be retrieved each time the probe or the composition changes. This can be done either by deleting an individual material from probe (using del probe[material]) or by clearing the entire cache.

```python
sld()

class refl1d.material.ProbeCache(probe=None)
    Probe proxy for materials properties.
    A caching probe which only looks up scattering factors for materials which it hasn’t seen before. Note that caching is based on object id, and will fail if the material object is updated with a new atomic structure.
    `probe` is the probe to use when looking up the scattering length density.
    The scattering factors need to be retrieved each time the probe or the composition changes. This can be done either by deleting an individual material from probe (using del probe[material]) or by clearing the entire cache.
```

### `scattering_factors(material, density)`

Return the scattering factors for the material, retrieving them from the cache if they have already been looked up.

### 4.13 `materialdb` - Materials Database

<table>
<thead>
<tr>
<th>material</th>
<th>Description of a solid block of material.</th>
</tr>
</thead>
<tbody>
<tr>
<td>air</td>
<td>Empty layer</td>
</tr>
<tr>
<td>water</td>
<td>Description of a solid block of material.</td>
</tr>
<tr>
<td>H2O</td>
<td>Description of a solid block of material.</td>
</tr>
<tr>
<td>heavywater</td>
<td>Description of a solid block of material.</td>
</tr>
<tr>
<td>D2O</td>
<td>Description of a solid block of material.</td>
</tr>
<tr>
<td>lightheavywater</td>
<td>Description of a solid block of material.</td>
</tr>
<tr>
<td>DHO</td>
<td>Description of a solid block of material.</td>
</tr>
<tr>
<td>silicon</td>
<td>Description of a solid block of material.</td>
</tr>
<tr>
<td>Si</td>
<td>Description of a solid block of material.</td>
</tr>
<tr>
<td>sapphire</td>
<td>Description of a solid block of material.</td>
</tr>
<tr>
<td>Al2O3</td>
<td>Description of a solid block of material.</td>
</tr>
<tr>
<td>gold</td>
<td>Description of a solid block of material.</td>
</tr>
<tr>
<td>Au</td>
<td>Description of a solid block of material.</td>
</tr>
<tr>
<td>permalloy</td>
<td>Description of a solid block of material.</td>
</tr>
<tr>
<td>Ni8Fe2</td>
<td>Description of a solid block of material.</td>
</tr>
</tbody>
</table>

Common materials in reflectometry experiments along with densities.
By name:

air, water, heavywater, lightheavywater, silicon, sapphire, gold
permalloy

By formula:

H2O, D2O, DHO, Si, Al2O3, Au, Ni8Fe2

If you want to adjust the density you will need to make your own copy of these materials. For example, for permalloy:

```python
>>> NiFe=Material(permalloy.formula,density=permalloy.bulk_density)
>>> NiFe.density.pmp(10)  # Let density vary by 10% from bulk value
Parameter(permalloy_density)
```

### 4.14 model - Reflectivity Models

<table>
<thead>
<tr>
<th>Repeat</th>
<th>Repeat a layer or stack.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slab</td>
<td>A block of material.</td>
</tr>
<tr>
<td>Stack</td>
<td>Reflectometry layer stack</td>
</tr>
<tr>
<td>Layer</td>
<td>Component of a material description.</td>
</tr>
</tbody>
</table>

Reflectometry models consist of 1-D stacks of layers. Layers are joined by gaussian interfaces. The layers themselves may be uniform, or the scattering density may vary with depth in the layer.

**Note:** By importing model, the definition of `material.Scatterer` changes so that materials can be stacked into layers using operator overloading: - the `|` operator, (previously known as “bitwise or”) joins stacks - the `*` operator repeats stacks (n times, n is an int)

This will affect all instances of the Scatterer class, and all of its subclasses.

```python
class refl1d.model.Repeat (stack, repeat=1, interface=None, name=None, magnetism=None)
```

**Bases:** `refl1d.model.Layer`

Repeat a layer or stack.

If an interface parameter is provide, the roughness between the multilayers may be different from the roughness between the repeated stack and the following layer.

**Note:** Repeat is not a type of Stack, but it does have a stack inside.

```python
constraints ()
```

Constraints

```python
find (z)
```

Find the layer at depth z.

Returns layer, start, end

```python
interface = None
```

```python
ismagnetic
```

```python
layer_parameters ()
```

---

4.14. model - Reflectivity Models 81
magnetism
parameters()
penalty()
render (probe, slabs)
thickness
class refl1d.model.Slab (material=None, thickness=0, interface=0, name=None, magnetism=None)
Bases: refl1d.model.Layer
A block of material.
constraints ()
    Constraints
find(z)
    Find the layer at depth z.
    Returns layer, start, end
interface = None
ismagnetic
layer_parameters ()
magnetism
parameters()
penalty()
    Return a penalty value associated with the layer. This should be zero if the parameters are valid, and
increasing as the parameters become more invalid. For example, if total volume fraction exceeds unity,
then the penalty would be the amount by which it exceeds unity, or if z values must be sorted, then penalty
would be the amount by which they are unsorted.

    Note that penalties are handled separately from any probability of seeing a combination of layer param-
eters; the final solution to the problem should not include any penalized points.
render (probe, slabs)
thickness = None
class refl1d.model.Stack (base=None, name='Stack')
Bases: refl1d.model.Layer
Reflectometry layer stack
A reflectometry sample is defined by a stack of layers. Each layer has an interface describing how the top of the
layer interacts with the bottom of the overlying layer. The stack may contain
add (other)
constraints ()
    Constraints
find(z)
    Find the layer at depth z.
    Returns layer, start, end
insert (idx, other)
    Insert structure into a stack. If the inserted element is another stack, the stack will be expanded to accom-
modate. You cannot make nested stacks.
interface = None
ismagnetic
layer_parameters()
magnetism
parameters()
penalty()
render (probe, slabs)
    Use the probe to render the layer into a microslab representation.

thickness
class refl1d.model.Layer
    Bases: object
    Component of a material description.

    thickness (Parameter: angstrom)    Thickness of the layer
    interface (Parameter: angstrom)    Interface for the top of the layer.
    magnetism (Magnetism info)        Magnetic profile anchored to the layer.

    constraints ()
        Constraints

    find(z)
        Find the layer at depth z.
        Returns layer, start, end

interface = None
ismagnetic
layer_parameters()
magnetism
parameters()

    Returns a dictionary of parameters specific to the layer. These will be added to the dictionary containing
    interface, thickness and magnetism parameters.

penalty()
    Return a penalty value associated with the layer. This should be zero if the parameters are valid, and
    increasing as the parameters become more invalid. For example, if total volume fraction exceeds unity,
    then the penalty would be the amount by which it exceeds unity, or if z values must be sorted, then penalty
    would be the amount by which they are unsorted.

    Note that penalties are handled separately from any probability of seeing a combination of layer parame-
    ters; the final solution to the problem should not include any penalized points.

render (probe, slabs)
    Use the probe to render the layer into a microslab representation.

thickness = None

4.15 mono - Freeform - Monotonic Spline
Monotonic spline modeling for free interfaces

class reflld.mono.FreeInterface (thickness=0, interface=0, below=None, above=None, dz=None, dp=None, name='Interface')

Bases: reflld.model.Layer

A freeform section of the sample modeled with monotonic splines.

Layers have a slope of zero at the ends, so the automatically blend with slabs.

constraints ()

Constraints

find(z)

Find the layer at depth z.

Returns layer, start, end

interface = None

ismagnetic

layer_parameters ()

magnetism

parameters ()

penalty ()

Return a penalty value associated with the layer. This should be zero if the parameters are valid, and increasing as the parameters become more invalid. For example, if total volume fraction exceeds unity, then the penalty would be the amount by which it exceeds unity, or if z values must be sorted, then penalty would be the amount by which they are unsorted.

Note that penalties are handled separately from any probability of seeing a combination of layer parameters; the final solution to the problem should not include any penalized points.

profile (Pz)

render (probe, slabs)

thickness = None

class reflld.mono.FreeLayer (below=None, above=None, thickness=0, z=[], rho=[], irho=[], name='Freeform')

Bases: reflld.model.Layer

A freeform section of the sample modeled with splines.

sld (rho) and imaginary sld (irho) can be modeled with a separate number of control points. The control points can be equally spaced in the layers unless rhoz or irhoz are specified. If the z values are given, they must be in the range [0,1]. One control point is anchored at either end, so there are two fewer z values than controls if z values are given.

Layers have a slope of zero at the ends, so the automatically blend with slabs.

constraints ()

Constraints
`find(z)`
- Find the layer at depth z.
- Returns layer, start, end

`interface = None`

`ismagnetic`

`layer_parameters()`

`magnetism parameters()`

`penalty()`

`profile(Pz, below, above)`

`render(probe, slabs)`

`thickness = None`

`refl1d.mono.inflections(dx, dy)`

### 4.16 names - Public API

---

**ModelFunction**

Exported names

In model definition scripts, rather than importing symbols one by one, you can simply perform:

```
from numpy import *
from refl1d.names import *
```

This is bad style for library and applications but convenient for small scripts.

```
refl1d.names.ModelFunction(*args, **kw)
```

### 4.17 ncnrdata - NCNR Data

<table>
<thead>
<tr>
<th>Instrument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANDR</td>
<td>Instrument definition for NCNR AND/R diffractometer/reflectometer.</td>
</tr>
<tr>
<td>MAGIK</td>
<td>Instrument definition for NCNR MAGIK diffractometer/reflectometer.</td>
</tr>
<tr>
<td>NCNRData</td>
<td>Instrument definition for NCNR NG-1 reflectometer.</td>
</tr>
<tr>
<td>NG1</td>
<td>Instrument definition for NCNR NG-1 reflectometer.</td>
</tr>
<tr>
<td>NG7</td>
<td>Instrument definition for NCNR NG-7 reflectometer.</td>
</tr>
<tr>
<td>PBR</td>
<td>Instrument definition for NCNR PBR reflectometer.</td>
</tr>
<tr>
<td>XRay</td>
<td>Instrument definition for NCNR X-ray reflectometer.</td>
</tr>
<tr>
<td>find_xsec</td>
<td>Find files containing the polarization cross-sections.</td>
</tr>
<tr>
<td>load</td>
<td>Return a probe for NCNR data.</td>
</tr>
<tr>
<td>load_magnetic</td>
<td>Return a probe for magnetic NCNR data.</td>
</tr>
<tr>
<td>parse_ncnr_file</td>
<td>Parse NCNR reduced data file returning header and data.</td>
</tr>
</tbody>
</table>

NCNR data loaders
The following instruments are defined:

MAGIK, PBR, ANDR, NG1, NG7 and XRay

These are `refl1d.instrument.Monochromatic` classes tuned with default instrument parameters and loaders for reduced NCNR data.

The instruments can be used to load data or to compute resolution functions for the purposes.

Example loading data:

```python
from refl1d.names import *

datafile = sample_data('chale207.refl')

instrument = NCNR.ANDR(Tlo=0.5, slits_at_Tlo=0.2, slits_below=0.1)

probe = instrument.load(datafile)

probe.plot(view='log')

Magnetic data has multiple cross sections and often has fixed slits:

```python
from refl1d.names import *

datafile = sample_data('lha03_255G.refl')

instrument = NCNR.NG1(slits_at_Tlo=1)

probe = instrument.load_magnetic(datafile)

probe.plot(view='SA', substrate=silicon) # Spin asymmetry view
```

For simulation, you need a probe and a sample:

```python
from refl1d.names import *

datafile = sample_data('lha03_255G.refl')

instrument = NCNR.ANDR(Tlo=0.5, slits_at_Tlo=0.2, slits_below=0.1)

probe = instrument.load(datafile)

probe.plot(view='SA', substrate=silicon) # Spin asymmetry view

M = Experiment(probe=probe, sample=sample)

M.simulate_data() # Optional

M.plot()
```

And for magnetic:

```python
from refl1d.names import *

instrument = NCNR.NG1(slits_at_Tlo=1)

M = Experiment(probe=probe, sample=sample)

M.simulate_data() # Optional

M.plot()

M = Experiment(probe=probe, sample=sample)

M.simulate_data() # Optional

M.plot()
```

See `instrument` for details.

class refl1d.ncnrdata.ANDR(**kw)
Bases: refl1d.ncnrdata.NCNRDaten, refl1d.instrument.Monochromatic

Instrument definition for NCNR AND/R diffractometer/reflectometer.

```python
Thi = 90

Tlo = 90

calc_dT(**kw)
Compute the angular divergence for given slits and angles

Parameters

T OR Q [float] [° | Å⁻¹] measurement angles
```
slits [float OR (float, float) | mm] total slit opening from edge to edge, not beam center to edge

d_s1, d_s2 [float | mm] distance from sample to slit 1 and slit 2

sample_width [float | mm] size of sample

sample_broadening [float | ° FWHM] resolution changes from sample warp

Returns

dT [[float] | ° FWHM] angular divergence

sample_broadening can be estimated from W, the full width at half maximum of a rocking curve measured in degrees:

\[
\text{sample_broadening} = W - \text{degrees} \left( \frac{0.5*(s1+s2)}{d1-d2} \right)
\]

calc_slits (**kw)
Determine slit openings from measurement pattern.

If slits are fixed simply return the same slits for every angle, otherwise use an opening range [Tlo, Thi] and
the value of the slits at the start of the opening to define the slits. Slits below Tlo and above Thi can be
specified separately.

T OR Q incident angle or Q Tlo, Thi angle range over which slits are opening slits_at_Tlo openings at the
start of the range, or fixed opening slits_below, slits_above openings below and above the range

Use fixed_slits is available, otherwise use opening slits.

dLoL = 0.009

d_s1 = 2086.0

d_s2 = 230.0

defaults ()
Return default instrument properties as a printable string.

fixed_slits = None

instrument = ‘AND/R’

load (filename, **kw)

load_magnetic (filename, **kw)

magnetic_probe (Aguide=270, shared_beam=True, H=0, **kw)
Simulate a polarized measurement probe.

Returns a probe with Q, angle, wavelength and the associated uncertainties, but not any data.

Guide field angle Aguide can be specified, as well as keyword arguments for the geometry of the probe
cross sections such as slits_at_Tlo, Tlo, Thi, slits_below, and slits_above to define the angular divergence.

probe (**kw)
Return a probe for use in simulation.

Parameters

Q [[float] | Å] Q values to be measured.

T [[float] | °] Angles to be measured.

Additional keyword parameters

Returns
probe  [Probe] Measurement probe with complete resolution information. The probe will not have any data.

If both $Q$ and $T$ are specified then $Q$ takes precedents.

You can override instrument parameters using key=value. In particular, settings for slits at Tlo, Tlo, Thi, slits below, and slits above are used to define the angular divergence.

radiation = ‘neutron’
readfile (filename)
resolution (**kw)
Calculate resolution at each angle.

Return


sample_broadening = 0
sample_width = 10000000000.0
slits_above = None
slits_at_Tlo = None
slits_below = None
wavelength = 5.0042

class refl1d.ncnrdata.MAGIK (**kw)
Bases: refl1d.ncnrdata.NCNRData, refl1d.instrument.Monochromatic
Instrument definition for NCNR MAGIK diffractometer/reflectometer.

Thi = 90
Tlo = 90
calc_dT (**kw)
Compute the angular divergence for given slits and angles

Parameters

$T$ OR $Q$ [[float] | ° OR Å$^{-1}$] measurement angles

slits [float OR (float,float) | mm] total slit opening from edge to edge, not beam center to edge
d_s1, d_s2 [float | mm] distance from sample to slit 1 and slit 2
sample_width [float | mm] size of sample
sample_broadening [float | ° FWHM] resolution changes from sample warp

Returns

d$T$ [[float] | ° FWHM] angular divergence

sample_broadening can be estimated from W, the full width at half maximum of a rocking curve measured in degrees:

sample_broadening = W - degrees(0.5*(s1+s2) / (d1-d2))
calc_slits (**kw)
Determines slit openings from measurement pattern.
If slits are fixed simply return the same slits for every angle, otherwise use an opening range [Tlo, Thi] and the value of the slits at the start of the opening to define the slits. Slits below Tlo and above Thi can be specified separately.

T OR Q incident angle or Q Tlo, Thi angle range over which slits are opening slits_at_Tlo openings at the start of the range, or fixed opening slits_below, slits_above openings below and above the range
Use fixed_slits is available, otherwise use opening slits.

dLoL = 0.009
d_s1 = 1759.0
d_s2 = 330.0
defaults ()
Return default instrument properties as a printable string.

fixed_slits = None
instrument = ‘MAGIK’
load (filename, **kw)
load_magnetic (filename, **kw)
magnetic_probe (Aguide=270, shared_beam=True, H=0, **kw)
Simulate a polarized measurement probe.
Returns a probe with Q, angle, wavelength and the associated uncertainties, but not any data.
Guide field angle Aguide can be specified, as well as keyword arguments for the geometry of the probe cross sections such as slits_at_Tlo, Tlo, Thi, slits_below, and slits_above to define the angular divergence.

probe (**kw)
Return a probe for use in simulation.

Parameters

Q [[float] | Å] Q values to be measured.
T [[float] | °] Angles to be measured.

Additional keyword parameters

Returns

probe [Probe] Measurement probe with complete resolution information. The probe will not have any data.
If both Q and T are specified then Q takes precedents.
You can override instrument parameters using key=value. In particular, settings for slits_at_Tlo, Tlo, Thi, slits_below, and slits_above are used to define the angular divergence.

radiation = ‘neutron’
readfile (filename)
resolution (**kw)
Calculate resolution at each angle.

Return

\[ L, dL \] Wavelengths and wavelength dispersion.

```python
sample_broadening = 0
sample_width = 10000000000.0
slits_above = None
slits_at_Tlo = None
slits_below = None
wavelength = 5.0042
```

class refl1d.ncnrdata.NCNRData

Bases: object

- `load(filename, **kw)`
- `load_magnetic(filename, **kw)`
- `readfile(filename)`

class refl1d.ncnrdata.NG1(**kw)

Bases: refl1d.ncnrdata.NCNRData, refl1d.instrument.Monochromatic

Instrument definition for NCNR NG-1 reflectometer.

\[ \text{Thi} = 90 \]

\[ \text{Tlo} = 90 \]

calc_dT(**kw)

Compute the angular divergence for given slits and angles

**Parameters**

- `T OR Q` ([float] \(\circ\) OR Å\(^{-1}\)) measurement angles
- `slits` [float OR (float, float) \(\text{mm}\)] total slit opening from edge to edge, not beam center to edge
- `d_s1, d_s2` [float \(\text{mm}\)] distance from sample to slit 1 and slit 2
- `sample_width` [float \(\text{mm}\)] size of sample
- `sample_broadening` [float \(\circ\) FWHM] resolution changes from sample warp

**Returns**

- `dT` [float \(\circ\) FWHM] angular divergence

`sample_broadening` can be estimated from \(W\), the full width at half maximum of a rocking curve measured in degrees:

\[ \text{sample_broadening} = \frac{W}{\text{degrees}(0.5\times(s1+s2)/(d1-d2))} \]

calc_slits(**kw)

Determines slit openings from measurement pattern.

If slits are fixed simply return the same slits for every angle, otherwise use an opening range \([Tlo, Thi]\) and the value of the slits at the start of the opening to define the slits. Slits below \(Tlo\) and above \(Thi\) can be specified separately.

\(T\ OR Q\) incident angle or \(Q\ Tlo, Thi\) angle range over which slits are opening `slits_at_Tlo` openings at the start of the range, or fixed opening `slits_below`, `slits_above` openings below and above the range

Use fixed_slits is available, otherwise use opening slits.
dLoL = 0.015

d_s1 = 1905.0

d_s2 = 355.59999999999997

d_s3 = 228.6

d_s4 = 1066.8

defaults()
    Return default instrument properties as a printable string.

fixed_slits = None

instrument = ‘NG-1’

load(filename, **kw)

load_magnetic(filename, **kw)

magnetic_probe(Aguide=270, shared_beam=True, H=0, **kw)
    Simulate a polarized measurement probe.

    Returns a probe with Q, angle, wavelength and the associated uncertainties, but not any data.

    Guide field angle Aguide can be specified, as well as keyword arguments for the geometry of the probe cross sections such as slits_at_Tlo, Tlo, Thi, slits_below, and slits_above to define the angular divergence.

probe(**kw)
    Return a probe for use in simulation.

    Parameters

        Q  [[float] | Å] Q values to be measured.
        T  [[float] | °] Angles to be measured.

    Additional keyword parameters

    Returns

        probe  [Probe] Measurement probe with complete resolution information. The probe will not have any data.

        If both Q and T are specified then Q takes precedents.

        You can override instrument parameters using key=value. In particular, settings for slits_at_Tlo, Tlo, Thi, slits_below, and slits_above are used to define the angular divergence.

radiation = ‘neutron’

readfile(filename)

resolution(**kw)
    Calculate resolution at each angle.

    Return


sample_broadening = 0

sample_width = 10000000000.0

slits_above = None
slits_at_Tlo = None
slits_below = None
wavelength = 4.75

class refl1d.ncnrdata.NG7(**kw)

Instrument definition for NCNR NG-7 reflectometer.

Thi = 90
Tlo = 90
calc_dT(**kw)

Compute the angular divergence for given slits and angles

Parameters

  T OR Q  [[float] | ° OR Å⁻¹] measurement angles
  slits  [float OR (float,float) | mm] total slit opening from edge to edge, not beam center to edge
  d_s1, d_s2 [float | mm] distance from sample to slit 1 and slit 2
  sample_width [float | mm] size of sample
  sample_broadening [float | ° FWHM] resolution changes from sample warp

Returns

  dT  [[float] | ° FWHM] angular divergence

  sample_broadening can be estimated from W, the full width at half maximum of a rocking curve measured in degrees:

  sample_broadening = W - degrees( 0.5*(s1+s2) / (d1-d2))

calc_slits(**kw)

Determines slit openings from measurement pattern.

If slits are fixed simply return the same slits for every angle, otherwise use an opening range [Tlo,Thi] and the value of the slits at the start of the opening to define the slits. Slits below Tlo and above Thi can be specified separately.

  T OR Q  incident angle or Q Tlo, Thi angle range over which slits are opening slits_at_Tlo openings at the start of the range, or fixed opening slits_below, slits_above openings below and above the range

Use fixed_slits is available, otherwise use opening slits.

dLoL = 0.025
d_detector = 2000.0
d_s1 = 1625.0
d_s2 = 275.0
defaults()

Return default instrument properties as a printable string.

fixed_slits = None
instrument = ‘NG-7’
load(filename, **kw)
**load_magnetic** *(filename, **kw)*

**magnetic_probe** *(Aguide=270, shared_beam=True, H=0, **kw)*

Simulate a polarized measurement probe.

- Returns a probe with Q, angle, wavelength and the associated uncertainties, but not any data.
- Guide field angle *Aguide* can be specified, as well as keyword arguments for the geometry of the probe cross sections such as *slits_at_Tlo*, *Tlo*, *Thi*, *slits_below*, and *slits_above* to define the angular divergence.

**probe** (**kw**)

Return a probe for use in simulation.

**Parameters**

- \( Q \) [[float | Å]] Q values to be measured.
- \( T \) [[float | °]] Angles to be measured.

**Returns**

- **probe** [Probe] Measurement probe with complete resolution information. The probe will not have any data.

If both \( Q \) and \( T \) are specified then \( Q \) takes precedents.

You can override instrument parameters using key=value. In particular, settings for *slits_at_Tlo*, *Tlo*, *Thi*, *slits_below*, and *slits_above* are used to define the angular divergence.

**radiation** = ‘neutron’

**readfile** *(filename)*

**resolution** (**kw**)

Calculate resolution at each angle.

**Return**

- \( T, dT \) [[float | °]] Angles and angular divergence.
- \( L, dL \) [[float | Å]] Wavelengths and wavelength dispersion.

**sample_broadening** = 0

**sample_width** = 10000000000.0

**slits_above** = None

**slits_at_Tlo** = None

**slits_below** = None

**wavelength** = 4.768

**class** refl1d.ncnrdata.PBR(**kw**)

**Bases:** refl1d.ncnrdata.NCNRData, refl1d.instrument.Monochromatic

Instrument definition for NCNR PBR reflectometer.

**Thi** = 90

**Tlo** = 90

**calc_dT** (**kw**)

Compute the angular divergence for given slits and angles

**Parameters**
**T OR Q** [[float] | ° OR Å⁻¹] measurement angles

**slits** [float OR (float,float) | mm] total slit opening from edge to edge, not beam center to edge

**d_s1, d_s2** [float | mm] distance from sample to slit 1 and slit 2

**sample_width** [float | mm] size of sample

**sample_broadening** [float | ° FWHM] resolution changes from sample warp

Returns

**dT** [[float] | ° FWHM] angular divergence

```
sample_broadening = W - degrees( 0.5*(s1+s2) / (d1-d2))
```

**calc_slits** (**kw**)

Determines slit openings from measurement pattern.

If slits are fixed simply return the same slits for every angle, otherwise use an opening range [Tlo,Thi] and the value of the slits at the start of the opening to define the slits. Slits below Tlo and above Thi can be specified separately.

**T OR Q** incident angle or Q **Tlo, Thi** angle range over which slits are opening **slits_at_Tlo** openings at the start of the range, or fixed opening **slits_below**, **slits_above** openings below and above the range

Use fixed_slits is available, otherwise use opening slits.

**dLoL = 0.015**

**d_s1 = 1835**

**d_s2 = 343**

**d_s3 = 380**

**d_s4 = 1015**

**defaults()**

Return default instrument properties as a printable string.

**fixed_slits = None**

**instrument = ‘PBR’**

**load** (filename, **kw**)

**load_magnetic** (filename, **kw**)

**magnetic_probe** (Aguide=270, shared_beam=True, H=0, **kw**)

Simulate a polarized measurement probe.

```
  Returns a probe with Q, angle, wavelength and the associated uncertainties, but not any data.
  Guide field angle Aguide can be specified, as well as keyword arguments for the geometry of the probe
cross sections such as slits_at_Tlo, Tlo, Thi, slits_below, and slits_above to define the angular divergence.
```

**probe** (**kw**)

Return a probe for use in simulation.

**Parameters**

**Q** [[float] | Å] Q values to be measured.
Refl1D: Neutron and X-Ray Reflectivity Analysis, Release 0.7.8

\[ T \] [[float] \[°\]] Angles to be measured.

Additional keyword parameters

Returns

probe  [Probe] Measurement probe with complete resolution information. The probe will not have any data.

If both \( Q \) and \( T \) are specified then \( Q \) takes precedents.

You can override instrument parameters using key=value. In particular, settings for \textit{slits\_at\_Tlo, Tlo, Thi, slits\_below, and slits\_above} are used to define the angular divergence.

\texttt{radiation = 'neutron'}
\texttt{readfile (filename)}
\texttt{resolution (**kw)}

Calculate resolution at each angle.

Return

\[ T, dT \] [[float] \[°\]] Angles and angular divergence.

\[ L, dL \] [[float] \[Å\]] Wavelengths and wavelength dispersion.

\texttt{sample\_broadening = 0}
\texttt{sample\_width = 10000000000.0}
\texttt{slits\_above = None}
\texttt{slits\_at\_Tlo = None}
\texttt{slits\_below = None}
\texttt{wavelength = 4.75}

\texttt{class refl1d.ncnrdata.XRay (**kw)}

\texttt{Bases: refl1d.ncnrdata.NCNRData, refl1d.instrument.Monochromatic}

Instrument definition for NCNR X-ray reflectometer.

Normal \( dT \) is in the range 2e-5 to 3e-4.

Slits are fixed throughout the experiment in one of a few preconfigured openings. Please update this file with the standard configurations when you find them.

You can choose to ignore the geometric calculation entirely by setting the slit opening to 0 and using \texttt{sample\_broadening} to define the entire divergence:

```
>>> from reflid.names import *
>>> file = sample_data("spin\_valve01.refl")
>>> xray = NCNR.XRay(slits\_at\_Tlo=0)
>>> data = xray.load(file, sample\_broadening=1e-4)
>>> print(data.dT[5])
0.0001

Thi = 90

Tlo = 90

\texttt{calc\_dT (**kw)}

Compute the angular divergence for given slits and angles

Parameters
**T OR Q** [[float] | ° OR Å⁻¹] measurement angles

**slits** [float OR (float,float) | mm] total slit opening from edge to edge, not beam center to edge

**d_s1, d_s2** [float | mm] distance from sample to slit 1 and slit 2

**sample_width** [float | mm] size of sample

**sample_broadening** [float | ° FWHM] resolution changes from sample warp

Returns

dT [[float] | ° FWHM] angular divergence

**sample_broadening** can be estimated from W, the full width at half maximum of a rocking curve measured in degrees:

\[
\text{sample_broadening} = W - \text{degrees}(0.5*(s1+s2)/(d1-d2))
\]

**calc_slits(**kw**)**

Determines slit openings from measurement pattern.

If slits are fixed simply return the same slits for every angle, otherwise use an opening range [Tlo,Thi] and the value of the slits at the start of the opening to define the slits. Slits below Tlo and above Thi can be specified separately.

**T OR Q** incident angle or Q 

**Tlo, Thi** angle range over which slits are opening 

**slits_at_Tlo** openings at the start of the range, or fixed opening 

**slits_below** , **slits_above** openings below and above the range

Use fixed_slits is available, otherwise use opening slits.

dLoL = 0.0006486766995329528

d_detector = None

d_s1 = 275.5

d_s2 = 192.5

d_s3 = 175.0

defaults()

Return default instrument properties as a printable string.

**fixed_slits** = None

**instrument** = ‘X-ray’

**load**(filename, **kw**)

**load_magnetic**(filename, **kw**)

**magnetic_probe**(Aguide=270, shared_beam=True, H=0, **kw**)

Simulate a polarized measurement probe.

Returns a probe with Q, angle, wavelength and the associated uncertainties, but not any data.

Guide field angle Aguide can be specified, as well as keyword arguments for the geometry of the probe cross sections such as slits_at_Tlo, Tlo, Thi, slits_below, and slits_above to define the angular divergence.

**probe**(**kw**)

Return a probe for use in simulation.

**Parameters**

**Q** [[float] | Å] Q values to be measured.
\(T\) \([\text{[float]} \times ^\circ]\) Angles to be measured.

Additional keyword parameters

Returns

- **probe** [Probe] Measurement probe with complete resolution information. The probe will not have any data.

If both \(Q\) and \(T\) are specified then \(Q\) takes precedents.

You can override instrument parameters using key=value. In particular, settings for \(\text{slits\_at\_Tlo, Tlo, Thi, slits\_below, and slits\_above}\) are used to define the angular divergence.

\[\text{radiation} = \text{‘xray’}\]

\[\text{readfile}(\text{filename})\]

\[\text{resolution}(**\text{kw})\]

Calculate resolution at each angle.

\[\text{Return}\]

- \(T, dT\) \([\text{[float]} \times ^\circ]\) Angles and angular divergence.

- \(L, dL\) \([\text{[float]} \times \text{Å}]\) Wavelengths and wavelength dispersion.

\[\text{sample\_broadening} = 0\]

\[\text{sample\_width} = 10000000000.0\]

\[\text{slits\_above} = \text{None}\]

\[\text{slits\_at\_Tlo} = \text{None}\]

\[\text{slits\_below} = \text{None}\]

\[\text{wavelength} = 1.5416\]

\[\text{refl1d.ncnrdata.\text{find\_xsec}}(\text{filename})\]

Find files containing the polarization cross-sections.

Returns tuple with file names for ++, +-, -+, –, or None if the spin cross section does not exist.

\[\text{refl1d.ncnrdata.\text{load}}(\text{filename}, \text{instrument=}\text{None}, **\text{kw})\]

Return a probe for NCNR data.

Keyword arguments are as specified Monochromatic instruments.

\[\text{refl1d.ncnrdata.\text{load\_magnetic}}(\text{filename, Aguide=270, H=0, shared\_beam=}\text{True}, **\text{kw})\]

Return a probe for magnetic NCNR data.

\[\text{filename (string, or 4x string)}\]

If it is a string, then filenameA, filenameB, filenameC, filenameD, are the –, -+, +-, ++ cross sections, otherwise the individual cross sections should be the filename for the cross section or None if the cross section does not exist.

\[\text{Aguide (degrees)}\]

Angle of the guide field relative to the beam. 270 is the default.

\[\text{shared\_beam (True)}\]

Use false if beam parameters should be fit separately for the individual cross sections.

Other keyword arguments are for the individual cross section loaders as specified in \text{instrument.Monochromatic}.

The data sets should be the base filename with an additional character corresponding to the spin state:
"a" corresponds to spin --
"b" corresponds to spin +- 
"c" corresponds to spin +--
"d" corresponds to spin ++

Unfortunately the interpretation is a little more complicated than this as the data acquisition system assigns letter on the basis of flipper state rather than neutron spin state. Whether flipper on or off corresponds to spin up or down depends on whether the polarizer/analyzer is a supermirror in transmission or reflection mode, or in the case of 3He polarizers, whether the polarization is up or down.

For full control, specify filename as a list of files, with None for the missing cross sections.

refl1d.ncnrdata.parse_ncnr_file(filename)
Parse NCNR reduced data file returning header and data.

header dictionary of fields such as ‘data’, ‘title’, ‘instrument’ data 2D array of data

If ‘columns’ is present in header, it will be a list of the names of the columns. If ‘instrument’ is present in the header, the default instrument geometry will be specified.

Slit geometry is set to the default from the instrument if it is not available in the reduced file.

4.18 polymer - Polymer models

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolymerBrush</td>
<td>Polymer brushes in a solvent</td>
</tr>
<tr>
<td>PolymerMushroom</td>
<td>Polymer mushrooms in a solvent (volume profile)</td>
</tr>
<tr>
<td>EndTetheredPolymer</td>
<td>Polymer end-tethered to an interface in a solvent</td>
</tr>
<tr>
<td>VolumeProfile</td>
<td>Generic volume profile function</td>
</tr>
</tbody>
</table>

layer_thickness
Return the thickness of a layer given the microslab z points.

Layer models for polymer systems.

Analytic Self-consistent Field (SCF) Brush profile

Analytical Self-consistent Field (SCF) Mushroom Profile

Numerical Self-consistent Field (SCF) End-Tethered Polymer Profile

class refl1d.polymer.PolymerBrush(thickness=0, interface=0, name='brush', polymer=None, solvent=None, base_vf=None, base=None, length=None, power=None, sigma=None)

Bases: refl1d.model.Layer

Polymer brushes in a solvent

Parameters

thickness the thickness of the solvent layer

---

interface  the roughness of the solvent surface
polymer  the polymer material
solvent  the solvent material or vacuum
base_vf  volume fraction (%) of the polymer brush at the interface
base  the thickness of the brush interface (A)
length  the length of the brush above the interface (A)
power  the rate of brush thinning
sigma  rms brush roughness (A)

The materials can either use the scattering length density directly, such as PDMS = SLD(0.063, 0.00006) or they can use chemical composition and material density such as PDMS=Material("C2H6OSi",density=0.965).

These parameters combine in the following profile formula:

\[
V(z) = \begin{cases} 
V_o & \text{if } z = z_o \\
V_o(1 - ((z - z_o)/L)^p) & \text{if } z_o < z < z_o + L \\
0 & \text{if } z > = z_o + L
\end{cases}
\]

\[
V_\sigma(z) = V(z) * e^{-\frac{1}{2}(z/\sigma)^2}
\]

\[
\rho(z) = \rho_p V_\sigma(z) + \rho_s (1 - V_\sigma(z))
\]

where \(V_\sigma(z)\) is volume fraction convoluted with brush roughness \(\sigma\) and \(\rho(z)\) is the complex scattering length density of the profile.

constraints()
Constraints

find(z)
Find the layer at depth z.
Returns layer, start, end

interface = None

is_magnetic

layer_parameters()

magnetism

parameters()

penalty()
Return a penalty value associated with the layer. This should be zero if the parameters are valid, and increasing as the parameters become more invalid. For example, if total volume fraction exceeds unity, then the penalty would be the amount by which it exceeds unity, or if \(z\) values must be sorted, then penalty would be the amount by which they are unsorted.

Note that penalties are handled separately from any probability of seeing a combination of layer parameters; the final solution to the problem should not include any penalized points.

profile(z)
render(probe, slabs)
thickness = None
class refl1d.polymer.PolymerMushroom (thickness=0, interface=0, name=u'Mushroom', polymer=None, solvent=None, sigma=0, vf=0, delta=0)

Bases: refl1d.model.Layer

Polymer mushrooms in a solvent (volume profile)

Parameters

  \( \delta \) | real scalar  interaction parameter

  \( \nu_f \) | real scalar  not quite volume fraction (dimensionless grafting density)

  \( \sigma \) | real scalar  convolution roughness (Å)

Using analytical SCF methods for gaussian chains, which are scaled by the radius of gyration of the equivalent free polymer as an approximation to results of renormalization group methods.\(^3\)

Solutions are only strictly valid for \( \nu_f \ll 1 \).

constraints ()

Constraints

find (z)

Find the layer at depth \( z \).

Returns layer, start, end

interface = None

is_magnetic

layer_parameters ()

magnetism

parameters ()

penalty ()

Return a penalty value associated with the layer. This should be zero if the parameters are valid, and increasing as the parameters become more invalid. For example, if total volume fraction exceeds unity, then the penalty would be the amount by which it exceeds unity, or if \( z \) values must be sorted, then penalty would be the amount by which they are unsorted.

Note that penalties are handled separately from any probability of seeing a combination of layer parameters; the final solution to the problem should not include any penalized points.

profile (z)

render (probe, slabs)

thickness = None

class refl1d.polymer.EndTetheredPolymer (thickness=0, interface=0, name=u'EndTetheredPolymer', polymer=None, solvent=None, chi=0, chi_s=0, h_dry=None, l_lat=1, mn=None, m_lat=1, pdi=1)

Bases: refl1d.model.Layer

Polymer end-tethered to an interface in a solvent

Uses a numerical self-consistent field profile.\(^456\)

Parameters

  \( \chi \)  solvent interaction parameter

  \( \chi_s \)  surface interaction parameter
Refl1D: Neutron and X-Ray Reflectivity Analysis, Release 0.7.8

$h_dry$ thickness of the neat polymer layer
$l_{lat}$ real length per lattice site
$mn$ Number average molecular weight
$m_{lat}$ real mass per lattice segment
$pdi$ Dispersity (Polydispersity index)

$thickness$ Slab thickness should be greater than the contour length of the polymer

$interface$ should be zero

$material$ the polymer material

$solvent$ the solvent material

Previous layer should not have roughness! Use a spline to simulate it.

According to $^7$, $l_{lat}$ and $m_{lat}$ should be calculated by the formulas:

\begin{eqnarray}
    l_{\text{lat}} &=& \frac{a^2 m/l}{p_l} \\
    m_{\text{lat}} &=& \frac{(a m/l)^2}{p_l}
\end{eqnarray}

where $l$ is the real polymer’s bond length, $m$ is the real segment mass, and $a$ is the ratio between molecular weight and radius of gyration at theta conditions. The lattice persistence, $p_l$, is:

$$p_l = \frac{1 + 1/Z}{6} \frac{1 - 1/Z}{6}$$

with coordination number $Z = 6$ for a cubic lattice, $p_l = .233$.

$constraints()$

Constraints

$find(z)$

Find the layer at depth $z$.

Returns layer, start, end

$interface = None$

$ismagnetic$

$layer\_parameters()$

$magnetism$

$parameters()$

$penalty()$

Return a penalty value associated with the layer. This should be zero if the parameters are valid, and increasing as the parameters become more invalid. For example, if total volume fraction exceeds unity, then the penalty would be the amount by which it exceeds unity, or if $z$ values must be sorted, then penalty would be the amount by which they are unsorted.

Note that penalties are handled separately from any probability of seeing a combination of layer parameters; the final solution to the problem should not include any penalized points.

$profile(z)$

$render(probe, slabs)$


4.18. polymer - Polymer models

101
**thickness** = None

```python
class refl1d.polymer.VolumeProfile(thickness=0, interface=0, name=u'VolumeProfile', material=None, solvent=None, profile=None, **kw)
```

**Bases:** refl1d.model.Layer

Generic volume profile function

**Parameters**

- **thickness**: the thickness of the solvent layer
- **interface**: the roughness of the solvent surface
- **material**: the polymer material
- **solvent**: the solvent material
- **profile**: the profile function, suitably parameterized

The materials can either use the scattering length density directly, such as PDMS = SLD(0.063, 0.00006) or they can use chemical composition and material density such as PDMS=Material("C2H6OSi", density=0.965).

These parameters combine in the following profile formula:

\[
sld = material.sld \times profile + solvent.sld \times (1 - profile)
\]

The profile function takes a depth \(z\) and returns a density \(\rho\).

For volume profiles, the returned \(\rho\) should be the volume fraction of the material. For SLD profiles, \(\rho\) should be complex scattering length density of the material.

Fitting parameters are the available named arguments to the function. The first argument must be \(z\), which is the array of depths at which the profile is to be evaluated. It is guaranteed to be increasing, with step size \(2 \times z[0]\).

Initial values for the function parameters can be given using name=value. These values can be scalars or fitting parameters. The function will be called with the current parameter values as arguments. The layer thickness can be computed as :func:`layer_thickness`.

**constraints()**

Constraints

**find(z)**

Find the layer at depth \(z\).

Returns layer, start, end

**interface** = None

**ismagnetic**

**layer_parameters()**

**magnetism**

**parameters()**

**penalty()**

Return a penalty value associated with the layer. This should be zero if the parameters are valid, and increasing as the parameters become more invalid. For example, if total volume fraction exceeds unity, then the penalty would be the amount by which it exceeds unity, or if \(z\) values must be sorted, then penalty would be the amount by which they are unsorted.

Note that penalties are handled separately from any probability of seeing a combination of layer parameters; the final solution to the problem should not include any penalized points.

**render**(probe, slabs)
thickness = None

refl1d.polymer.layer_thickness(z)
Return the thickness of a layer given the microslab z points.

The z points are at the centers of the bins. we can use the recurrence that boundary b[k] = z[k-1] + (z[k-1] - b[k-1]) to compute the total length of the layer.

4.19 probe - Instrument probe

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NeutronProbe</td>
<td>Neutron probe.</td>
</tr>
<tr>
<td>PolarizedNeutronProbe</td>
<td>Polarized neutron probe</td>
</tr>
<tr>
<td>PolarizedNeutronQProbe</td>
<td>alias of PolarizedQProbe</td>
</tr>
<tr>
<td>PolarizedQProbe</td>
<td></td>
</tr>
<tr>
<td>Probe</td>
<td>Defines the incident beam used to study the material.</td>
</tr>
<tr>
<td>ProbeSet</td>
<td></td>
</tr>
<tr>
<td>QProbe</td>
<td>A pure Q, R probe</td>
</tr>
<tr>
<td>Qmeasurement_union</td>
<td>Determine the unique Q, dQ across all datasets.</td>
</tr>
<tr>
<td>XRayProbe</td>
<td>X-Ray probe.</td>
</tr>
<tr>
<td>load4</td>
<td>Load in four column data Q, R, dR, dQ.</td>
</tr>
<tr>
<td>make_probe</td>
<td>Return a reflectometry measurement object of the given resolution.</td>
</tr>
<tr>
<td>measurement_union</td>
<td>Determine the unique (T, dT, L, dL) across all datasets.</td>
</tr>
<tr>
<td>spin_asymmetry</td>
<td>Compute spin asymmetry for R++, R–.</td>
</tr>
</tbody>
</table>

Experimental probe.

The experimental probe describes the incoming beam for the experiment. Scattering properties of the sample are dependent on the type and energy of the radiation.

See Data Representation for details.

class refl1d.probe.NeutronProbe(T=None, dT=0, L=None, dL=0, data=None, intensity=1, background=0, back_absorption=1, theta_offset=0, back_reflectivity=False, name=None, filename=None)

Bases: refl1d.probe.Probe

Neutron probe.

By providing a scattering factor calculator for X-ray scattering, model components can be defined by mass density and chemical composition.

Aguide = 270

Q

alignment_uncertainty(w, I, d=0)
Compute alignment uncertainty.

Parameters:

w [float | degrees] Rocking curve full width at half max.
I [float | counts] Rocking curve integrated intensity.

d = 0: float | degrees Motor step size

Returns:

dtheta [float | degrees] uncertainty in alignment angle
apply_beam(calc_Q, calc_R, resolution=True, interpolation=0)

Apply factors such as beam intensity, background, backabsorption, resolution to the data.

calc_Q

critical_edge(substrate=None, surface=None, n=51, delta=0.25)

Oversample points near the critical edge.

The critical edge is defined by the difference in scattering potential for the substrate and surface materials, or the reverse if back_reflectivity is true.

n is the number of Q points to compute near the critical edge.

delta is the relative uncertainty in the material density, which defines the range of values which are calculated.

The n points $Q_i$ are evenly distributed around the critical edge in $Q_c \pm \delta Q_c$ by varying angle $\theta$ for a fixed wavelength $<\lambda>$, the average of all wavelengths in the probe.

Specifically:

$$Q_c^2 = 16\pi(\rho - \rho_{\text{incident}})$$

$$Q_i = Q_c - \delta_i Q_c(i - (n - 1)/2) \quad \text{for } i \in 0 \ldots n - 1$$

$$\lambda_i = <\lambda>$$

$$\theta_i = \sin^{-1}(Q_i \lambda_i/4\pi)$$

If $Q_c$ is imaginary, then $-|Q_c|$ is used instead, so this routine can be used for reflectivity signals which scan from back reflectivity to front reflectivity. For completeness, the angle $\theta = 0$ is added as well.

fresnel(substrate=None, surface=None)

Returns a Fresnel reflectivity calculator given the surface and and substrate. The calculated reflectivity includes The Fresnel reflectivity for the probe reflecting from a block of material with the given substrate.

Returns $F = R(\text{probe}.Q)$, where $R$ is magnitude squared reflectivity.

label(prefix=None, gloss='', suffix='')

log10_to_linear()

Convert data from log to linear.

Older reflectometry reduction code stored reflectivity in log base 10 format. Call probe.log10_to_linear() after loading this data to convert it to linear for subsequent display and fitting.

oversample(n=20, seed=1)

Generate an over-sampling of Q to avoid aliasing effects.

Oversampling is needed for thick layers, in which the underlying reflectivity oscillates so rapidly in Q that a single measurement has contributions from multiple Kissig fringes.

Sampling will be done using a pseudo-random generator so that accidental structure in the function does not contribute to the aliasing. The generator will usually be initialized with a fixed seed so that the point selection will not change from run to run, but a seed of None will choose a different set of points each time oversample is called.

The value $n$ is the number of points that should contribute to each Q value when computing the resolution. These will be distributed about the nominal measurement value, but varying in both angle and energy according to the resolution function. This will yield more points near the measurement and fewer farther away. The measurement point itself will not be used to avoid accidental bias from uniform Q steps. Depending on the problem, a value of $n$ between 20 and 100 should lead to stable values for the convolved reflectivity.

parameters()
**plot** (view=\texttt{None}, **kwargs)
Plot theory against data.

Need substrate/surface for Fresnel-normalized reflectivity

**plot\_Q4** (**kwargs)
Plot the \(Q^4\) reflectivity associated with the probe.

Note that \(Q^4\) reflectivity has the intensity and background applied so that hydrogenated samples display more cleanly. The formula to reproduce the graph is:

\[
R' = R/(100 * Q)^4 I + B)
\]

\[
\Delta R' = \Delta R/(100 * Q)^4 I + B)
\]

where \(B\) is the background.

**plot\_fft** (theory=\texttt{None}, suffix=’, label=\texttt{None}, substrate=\texttt{None}, surface=\texttt{None}, **kwargs)
FFT analysis of reflectivity signal.

**plot\_fresnel** (substrate=\texttt{None}, surface=\texttt{None}, **kwargs)
Plot the Fresnel-normalized reflectivity associated with the probe.

Note that the Fresnel reflectivity has the intensity and background applied before normalizing so that hydrogenated samples display more cleanly. The formula to reproduce the graph is:

\[
R' = R/(F(Q)I + B)
\]

\[
\Delta R' = \Delta R/(F(Q)I + B)
\]

where \(I\) is the intensity and \(B\) is the background.

**plot\_linear** (**kwargs)
Plot the data associated with probe.

**plot\_log** (**kwargs)
Plot the data associated with probe.

**plot\_logfresnel** (*args, **kw)
Plot the log Fresnel-normalized reflectivity associated with the probe.

**plot\_residuals** (theory=\texttt{None}, suffix=’, label=\texttt{None}, plot\_shift=\texttt{None}, **kwargs)

**plot\_resolution** (suffix=’, label=\texttt{None}, **kwargs)

**plot\_shift** = \texttt{0}

**polarized** = \texttt{False}

**radiation** = ‘\texttt{neutron}’

**residuals\_shift** = \texttt{0}

**resolution\_guard**()
Make sure each measured \(Q\) point has at least 5 calculated \(Q\) points contributing to it in the range \([-3\Delta Q, 3\Delta Q]\).

*Not Implemented*

**restore\_data**()
Restore the original data.

**resynth\_data**()
Generate new data according to the model \(R \sim N(Ro, dR)\).

The resynthesis step is a precursor to refitting the data, as is required for certain types of monte carlo error analysis.

**save** (filename, theory, substrate=\texttt{None}, surface=\texttt{None})
Save the data and theory to a file.
scattering_factors (material, density)
Returns the scattering factors associated with the material given the range of wavelengths/energies used in the probe.

simulate_data (theory, noise=None)
Set the data for the probe to R, adding random noise dR.
If noise is None, then use the uncertainty in the probe.
As a hack, if noise<0, use the probe uncertainty but don’t add noise to the data. Don’t depend on this behavior.

subsample (dQ)
Select points at most every dQ.
Use this to speed up computation early in the fitting process.
This changes the data object, and is not reversible.
The current algorithm is not picking the “best” Q value, just the nearest, so if you have nearby Q points with different quality statistics (as happens in overlapped regions from spallation source measurements at different angles), then it may choose badly. Simple solutions based on the smallest relative error dR/R will be biased toward peaks, and smallest absolute error dR will be biased toward valleys.

view = ‘fresnel’

write_data (filename, columns=('Q', 'R', 'dR'), header=None)
Save the data to a file.
header is a string with trailing n containing the file header. columns is a list of column names from Q, dQ, R, dR, L, dL, T, dT.
The default is to write Q, R, dR data.

class refl1d.probe.PolarizedNeutronProbe (xs=None, name=None, Aguide=270, H=0)
Bases: object
Polarized neutron probe
xs (4 x NeutronProbe) is a sequence pp, pm, mp and mm.
Aguide (degrees) is the angle of the applied field relative to the plane of the sample, with angle 270° in the plane of the sample.
H (tesla) is the magnitude of the applied field
apply_beam (Q, R, resolution=True, interpolation=0)
Apply factors such as beam intensity, background, backabsorption, and footprint to the data.

calc_Q
fresnel (*args, **kw)
Returns a Fresnel reflectivity calculator given the surface and and substrate. The calculated reflectivity includes The Fresnel reflectivity for the probe reflecting from a block of material with the given substrate.
Returns F = R(probe.Q), where R is magnitude squared reflectivity.

mm
mp
oversample (n=6, seed=1)
Generate an over-sampling of Q to avoid aliasing effects.
Oversampling is needed for thick layers, in which the underlying reflectivity oscillates so rapidly in Q that a single measurement has contributions from multiple Kissig fringes.
Sampling will be done using a pseudo-random generator so that accidental structure in the function does not contribute to the aliasing. The generator will usually be initialized with a fixed seed so that the point selection will not change from run to run, but a seed of None will choose a different set of points each time oversample is called.

The value \( n \) is the number of points that should contribute to each Q value when computing the resolution. These will be distributed about the nominal measurement value, but varying in both angle and energy according to the resolution function. This will yield more points near the measurement and fewer farther away. The measurement point itself will not be used to avoid accidental bias from uniform Q steps. Depending on the problem, a value of \( n \) between 20 and 100 should lead to stable values for the convolved reflectivity.

```
parameters()

plot(view=None, **kwargs)
   Plot theory against data.
   Need substrate/surface for Fresnel-normalized reflectivity

plot_Q4(**kwargs)
plot_SA(theory=None, label=None, plot_shift=None, **kwargs)
plot_fresnel(**kwargs)
plot_linear(**kwargs)
plot_log(**kwargs)
plot_logfresnel(**kwargs)
plot_residuals(**kwargs)
plot_resolution(**kwargs)

pm
polarized = True

PP

restore_data()
   Restore the original data.

resynth_data()
   Generate new data according to the model \( R \sim N(R_0, dR) \).
   The resynthesis step is a precursor to refitting the data, as is required for certain types of monte carlo error analysis.

save(filename, theory, substrate=None, surface=None)
   Save the data and theory to a file.

scattering_factors(material, density)
   Returns the scattering factors associated with the material given the range of wavelengths/energies used in the probe.

select_corresponding(theory)
   Select theory points corresponding to the measured data.
   Since we have evaluated theory at every Q, it is safe to interpolate measured Q into theory, since it will land on a node, not in an interval.

shared_beam(intensity=1, background=0, back_absorption=1, theta_offset=0)
   Share beam parameters across all four cross sections.
```
New parameters are created for intensity, background, theta_offset and back_absorption and assigned to the all cross sections. These can be replaced in an individual cross section if for some reason one of the parameters is independent.

**simulate_data**(theory, noise=2)
Set the data for the probe to R, adding random noise dR.

If noise is None, then use the uncertainty in the probe.

As a hack, if noise<0, use the probe uncertainty but don’t add noise to the data. Don’t depend on this behavior.

**substrate = None**
**surface = None**
**view = None**

xs

refl1d.probe.PolarizedNeutronQProbe
alias of PolarizedQProbe
class refl1d.probe.PolarizedQProbe(xs=None, name=None, Aguide=270, H=0)
Bases: refl1d.probe.PolarizedNeutronProbe

**apply_beam**(Q, R, resolution=True, interpolation=0)
Apply factors such as beam intensity, background, backabsorption, and footprint to the data.

**calc_Q**

fresnel(*args, **kw)
Returns a Fresnel reflectivity calculator given the surface and and substrate. The calculated reflectivity includes The Fresnel reflectivity for the probe reflecting from a block of material with the given substrate.

Returns F = R(probe.Q), where R is magnitude squared reflectivity.

**oversample**(n=6, seed=1)
Generate an over-sampling of Q to avoid aliasing effects.

Oversampling is needed for thick layers, in which the underlying reflectivity oscillates so rapidly in Q that a single measurement has contributions from multiple Kissig fringes.

Sampling will be done using a pseudo-random generator so that accidental structure in the function does not contribute to the aliasing. The generator will usually be initialized with a fixed seed so that the point selection will not change from run to run, but a seed of None will choose a different set of points each time oversample is called.

The value n is the number of points that should contribute to each Q value when computing the resolution. These will be distributed about the nominal measurement value, but varying in both angle and energy according to the resolution function. This will yield more points near the measurement and fewer farther away. The measurement point itself will not be used to avoid accidental bias from uniform Q steps. Depending on the problem, a value of n between 20 and 100 should lead to stable values for the convolved reflectivity.

**parameters**()

**plot**(view=None, **kwargs)
Plot theory against data.

Need substrate/surface for Fresnel-normalized reflectivity
plot_Q4(**kwargs)

plot_SA(theory=None, label=None, plot_shift=None, **kwargs)

plot_fresnel(**kwargs)

plot_linear(**kwargs)

plot_log(**kwargs)

plot_logfresnel(**kwargs)

plot_residuals(**kwargs)

plot_resolution(**kwargs)

pm

polarized=True

PP

restore_data()

   Restore the original data.

resynth_data()

   Generate new data according to the model $R \sim N(R_0, dR)$.

   The resynthesis step is a precursor to refitting the data, as is required for certain types of monte carlo error
   analysis.

save(filename, theory, substrate=None, surface=None)

   Save the data and theory to a file.

scattering_factors(material, density)

   Returns the scattering factors associated with the material given the range of wavelengths/energies used in
   the probe.

select_corresponding(theory)

   Select theory points corresponding to the measured data.

   Since we have evaluated theory at every Q, it is safe to interpolate measured Q into theory, since it will
   land on a node, not in an interval.

shared_beam(intensity=1, background=0, back_absorption=1, theta_offset=0)

   Share beam parameters across all four cross sections.

   New parameters are created for intensity, background, theta_offset and back_absorption and assigned to
   the all cross sections. These can be replaced in an individual cross section if for some reason one of the
   parameters is independent.

simulate_data(theory, noise=2)

   Set the data for the probe to $R$, adding random noise $dR$.

   If noise is None, then use the uncertainty in the probe.

   As a hack, if noise<0, use the probe uncertainty but don’t add noise to the data. Don’t depend on this
   behavior.

substrate=None

surface=None

view=None

xs
class reflid.probe.Probe(T=None, dT=0, L=None, dL=0, data=None, intensity=1, background=0, back_absorption=1, theta_offset=0, back_reflectivity=False, name=None, filename=None)

Bases: object

Defines the incident beam used to study the material.

For calculation purposes, probe needs to return the values $Q_{\text{calc}}$ at which the model is evaluated. This is normally going to be the measured points only, but for some systems, such as those with very thick layers, oversampling is needed to avoid aliasing effects.

A measurement point consists of incident angle, angular resolution, incident wavelength, FWHM wavelength resolution, reflectivity and uncertainty in reflectivity.

A probe is a set of points, defined by vectors for point attribute. For convenience, the attribute can be initialized with a scalar if it is constant throughout the measurement, but will be set to a vector in the probe. The attributes are initialized as follows:

- $T$ [float or [float] | degrees] Incident angle
- $dT$ [float or [float] | degrees] FWHM angular resolution
- $L$ [float or [float] | Å] Incident wavelength
- $dL$ [float or [float] | Å] FWHM wavelength dispersion
- $data$ [[(float), [float]]] $R$, $dR$ reflectivity measurement and uncertainty

Measurement properties:

- $intensity$ [float or Parameter] Beam intensity
- $background$ [float or Parameter] Constant background
- $back\_absorption$ [float or Parameter] Absorption through the substrate relative to beam intensity. A value of 1.0 means complete transmission; a value of 0.0 means complete absorption.
- $theta\_offset$ [float or Parameter] Offset of the sample from perfect alignment
- $back\_reflectivity$ [True or False] True if the beam enters through the substrate

Measurement properties are fittable parameters. $theta\_offset$ in particular should be set using `probe.theta_offset.dev(dT)`, with $dT$ equal to the FWHM uncertainty in the peak position for the rocking curve, as measured in radians. Changes to $theta\_offset$ will then be penalized in the cost function for the fit as if it were another measurement. Use `alignment_uncertainty()` to compute $dT$ from the shape of the rocking curve.

intensity and back_absorption are generally not needed — scaling the reflected signal by an appropriate intensity measurement will correct for both of these during reduction. background may be needed, particularly for samples with significant hydrogen content due to its large isotropic incoherent scattering cross section.

View properties:

- `plot_shift` [float] The number of pixels to shift each new dataset so datasets can be seen separately
- `residuals_shift` [] The number of pixels to shift each new set of residuals so the residuals plots can be seen separately.

Normally view is set directly in the class rather than the instance since it is not specific to the view. Fresnel and Q4 views are corrected for background and intensity; log and linear views show the uncorrected data.

Aguide = 270
Q
**static alignment uncertainty** \((w, I, d=0)\)
Compute alignment uncertainty.

**Parameters:**
- \(w\) [float | degrees] Rocking curve full width at half max.
- \(I\) [float | counts] Rocking curve integrated intensity.
- \(d = 0\): float | degrees Motor step size

**Returns:**
- \(d\theta\) [float | degrees] uncertainty in alignment angle

**apply_beam** \((calc_Q, calc_R, resolution=True, interpolation=0)\)
Apply factors such as beam intensity, background, backabsorption, resolution to the data.

**calc_Q**

**critical_edge** \((substrate=None, surface=None, n=51, delta=0.25)\)
Oversample points near the critical edge.

The critical edge is defined by the difference in scattering potential for the `substrate` and `surface` materials, or the reverse if `back_reflectivity` is true.

\(n\) is the number of \(Q\) points to compute near the critical edge.

\(delta\) is the relative uncertainty in the material density, which defines the range of values which are calculated.

The \(n\) points \(Q_i\) are evenly distributed around the critical edge in \(Q_c \pm \delta Q_c\) by varying angle \(\theta\) for a fixed wavelength < \(\lambda >\), the average of all wavelengths in the probe.

Specifically:

\[
Q_c^2 = 16\pi(\rho - \rho_{\text{incident}}) \\
Q_i = Q_c - \delta Q_c(i - (n - 1)/2) \quad \text{for } i \in 0 \ldots n - 1 \\
\lambda_i = < \lambda > \\
\theta_i = \sin^{-1}(Q_i\lambda_i/4\pi)
\]

If \(Q_c\) is imaginary, then \(-|Q_c|\) is used instead, so this routine can be used for reflectivity signals which scan from back reflectivity to front reflectivity. For completeness, the angle \(\theta = 0\) is added as well.

**fresnel** \((substrate=None, surface=None)\)
Returns a Fresnel reflectivity calculator given the surface and substrate. The calculated reflectivity includes The Fresnel reflectivity for the probe reflecting from a block of material with the given substrate.

Returns \(F = R(\text{probe}.Q)\), where \(R\) is magnitude squared reflectivity.

**label** \((prefix=None, gloss='', suffix='')\)

**log10_to_linear**()
Convert data from log to linear.

Older reflectometry reduction code stored reflectivity in log base 10 format. Call `probe.log10_to_linear()` after loading this data to convert it to linear for subsequent display and fitting.

**oversample** \((n=20, seed=1)\)
Generate an over-sampling of \(Q\) to avoid aliasing effects.

Oversampling is needed for thick layers, in which the underlying reflectivity oscillates so rapidly in \(Q\) that a single measurement has contributions from multiple Kissig fringes.
Sampling will be done using a pseudo-random generator so that accidental structure in the function does not contribute to the aliasing. The generator will usually be initialized with a fixed seed so that the point selection will not change from run to run, but a seed of None will choose a different set of points each time oversample is called.

The value \( n \) is the number of points that should contribute to each \( Q \) value when computing the resolution. These will be distributed about the nominal measurement value, but varying in both angle and energy according to the resolution function. This will yield more points near the measurement and fewer farther away. The measurement point itself will not be used to avoid accidental bias from uniform \( Q \) steps. Depending on the problem, a value of \( n \) between 20 and 100 should lead to stable values for the convolved reflectivity.

**parameters()**

**plot**(\( \text{view=None, **kwargs} \))
Plot theory against data.

Need substrate/surface for Fresnel-normalized reflectivity

**plot_Q4**(\( **kwargs \))
Plot the \( Q^4 \) reflectivity associated with the probe.

Note that \( Q^4 \) reflectivity has the intensity and background applied so that hydrogenated samples display more cleanly. The formula to reproduce the graph is:

\[
R' = R/((100 \ast Q)^{-4}I + B) \\
\Delta R' = \Delta R/((100 \ast Q)^{-4}I + B)
\]

where \( B \) is the background.

**plot_fft**(\( \text{theory=None, suffix='', label=None, substrate=None, surface=None, **kwargs} \))
FFT analysis of reflectivity signal.

**plot_fresnel**(\( \text{substrate=None, surface=None, **kwargs} \))
Plot the Fresnel-normalized reflectivity associated with the probe.

Note that the Fresnel reflectivity has the intensity and background applied before normalizing so that hydrogenated samples display more cleanly. The formula to reproduce the graph is:

\[
R' = R/(F(Q)I + B) \\
\Delta R' = \Delta R/(F(Q)I + B)
\]

where \( I \) is the intensity and \( B \) is the background.

**plot_linear**(\( **kwargs \))
Plot the data associated with probe.

**plot_log**(\( **kwargs \))
Plot the data associated with probe.

**plot_logfresnel**(\( \text{args, **kw} \))
Plot the log Fresnel-normalized reflectivity associated with the probe.

**plot_residuals**(\( \text{theory=None, suffix='', label=None, plot_shift=None, **kwargs} \))

**plot_resolution**(\( \text{suffix='', label=None, **kwargs} \))

**plot_shift** = 0

**polarized** = False

**residuals_shift** = 0

**resolution_guard()**
Make sure each measured \( Q \) point has at least 5 calculated \( Q \) points contributing to it in the range \([-3\Delta Q, 3\Delta Q]\).
Not Implemented

**restore_data()**

Restore the original data.

**resynth_data()**

Generate new data according to the model $R \sim N(R_0, dR)$.

The resynthesis step is a precursor to refitting the data, as is required for certain types of monte carlo error analysis.

**save(filename, theory, substrate=None, surface=None)**

Save the data and theory to a file.

**scattering_factors(material, density)**

Returns the scattering factors associated with the material given the range of wavelengths/energies used in the probe.

**simulate_data(theory, noise=None)**

Set the data for the probe to $R$, adding random noise $dR$.

If noise is None, then use the uncertainty in the probe.

As a hack, if noise<0, use the probe uncertainty but don’t add noise to the data. Don’t depend on this behavior.

**subsample(dQ)**

Select points at most every $dQ$.

Use this to speed up computation early in the fitting process.

This changes the data object, and is not reversible.

The current algorithm is not picking the “best” $Q$ value, just the nearest, so if you have nearby $Q$ points with different quality statistics (as happens in overlapped regions from spallation source measurements at different angles), then it may choose badly. Simple solutions based on the smallest relative error $dR/R$ will be biased toward peaks, and smallest absolute error $dR$ will be biased toward valleys.

**view = ‘fresnel’**

**write_data(filename, columns=('Q', 'R', 'dR'), header=None)**

Save the data to a file.

*header* is a string with trailing n containing the file header. *columns* is a list of column names from Q, dQ, R, dR, L, dL, T, dT.

The default is to write Q, R, dR data.

**class refl1d.probe.ProbeSet(probes, name=None)**

Bases: refl1d.probe.Probe

**Aguide = 270**

**Q**

**alignment_uncertainty(w, I, d=0)**

Compute alignment uncertainty.

*Parameters:*

- $w$ [float | degrees] Rocking curve full width at half max.
- $I$ [float | counts] Rocking curve integrated intensity.
- $d = 0$: [float | degrees] Motor step size
Returns:

*dtheta* [float | degrees] uncertainty in alignment angle

**apply_beam**(calc_Q, calc_R, interpolation=0, **kw)**

**calc_Q**

**critical_edge**(substrate=None, surface=None, n=51, delta=0.25)

Oversample points near the critical edge.

The critical edge is defined by the difference in scattering potential for the substrate and surface materials, or the reverse if *back_reflectivity* is true.

*n* is the number of *Q* points to compute near the critical edge.

*delta* is the relative uncertainty in the material density, which defines the range of values which are calculated.

The *n* points *Q_i* are evenly distributed around the critical edge in *Q_c ± Q_i* by varying angle *θ* for a fixed wavelength < *λ*, the average of all wavelengths in the probe.

Specifically:

\[
Q_i^2 = 16\pi (\rho - \rho_{\text{incident}}) \\
Q_i = Q_c - \delta_i Q_c (i - (n - 1)/2) \quad \text{for } i \in 0 \ldots n - 1 \\
\lambda_i = \langle \lambda \rangle \\
\theta_i = \sin^{-1}(Q_i \lambda_i/4\pi) 
\]

If *Q_c* is imaginary, then −|*Q_c*| is used instead, so this routine can be used for reflectivity signals which scan from back reflectivity to front reflectivity. For completeness, the angle *θ* = 0 is added as well.

**fresnel**(*args, **kw**)

Returns a Fresnel reflectivity calculator given the surface and and substrate. The calculated reflectivity includes The Fresnel reflectivity for the probe reflecting from a block of material with the given substrate.

Returns *F = R(probe.Q)*, where *R* is magnitude squared reflectivity.

**label**(prefix=None, gloss='', suffix='')

**log10_to_linear**()

Convert data from log to linear.

Older reflectometry reduction code stored reflectivity in log base 10 format. Call probe.log10_to_linear() after loading this data to convert it to linear for subsequent display and fitting.

**oversample**(**kw**)

Generate an over-sampling of *Q* to avoid aliasing effects.

Oversampling is needed for thick layers, in which the underlying reflectivity oscillates so rapidly in *Q* that a single measurement has contributions from multiple Kissig fringes.

Sampling will be done using a pseudo-random generator so that accidental structure in the function does not contribute to the aliasing. The generator will usually be initialized with a fixed *seed* so that the point selection will not change from run to run, but a *seed* of None will choose a different set of points each time oversample is called.

The value *n* is the number of points that should contribute to each *Q* value when computing the resolution. These will be distributed about the nominal measurement value, but varying in both angle and energy according to the resolution function. This will yield more points near the measurement and fewer farther away. The measurement point itself will not be used to avoid accidental bias from uniform *Q* steps. Depending on the problem, a value of *n* between 20 and 100 should lead to stable values for the convolved reflectivity.
parameters ()

parts (theory)

plot (theory=None, **kw)
Plot theory against data.

Need substrate/surface for Fresnel-normalized reflectivity

plot_Q4 (theory=None, **kw)
Plot the Q^4 reflectivity associated with the probe.

Note that Q^4 reflectivity has the intensity and background applied so that hydrogenated samples display more cleanly. The formula to reproduce the graph is:

\[ R' = \frac{R}{((100 \cdot Q)^{-4}I + B)\Delta R'} = \frac{\Delta R}{((100 \cdot Q)^{-4}I + B)} \]

where \( B \) is the background.

plot_fft (theory=None, suffix='', label=None, substrate=None, surface=None, **kwargs)
FFT analysis of reflectivity signal.

plot_fresnel (theory=None, **kw)
Plot the Fresnel-normalized reflectivity associated with the probe.

Note that the Fresnel reflectivity has the intensity and background applied before normalizing so that hydrogenated samples display more cleanly. The formula to reproduce the graph is:

\[ R' = \frac{R}{(F(Q)I + B)\Delta R'} = \frac{\Delta R}{(F(Q)I + B)} \]

where \( I \) is the intensity and \( B \) is the background.

plot_linear (theory=None, **kw)
Plot the data associated with probe.

plot_log (theory=None, **kw)
Plot the data associated with probe.

plot_logfresnel (theory=None, **kw)
Plot the log Fresnel-normalized reflectivity associated with the probe.

plot_residuals (theory=None, **kw)
plot_resolution (**kw)
plot_shift = 0
polarized = False
residuals_shift = 0
resolution_guard()
Make sure each measured \( Q \) point has at least 5 calculated \( Q \) points contributing to it in the range \([-3\Delta Q, 3\Delta Q]\).

Not Implemented

restore_data ()
Restore the original data.

resynth_data ()
Generate new data according to the model \( R \sim N(R_0, dR) \).

The resynthesis step is a precursor to refitting the data, as is required for certain types of monte carlo error analysis.
**save** *(filename, theory, substrate=None, surface=None)*

Save the data and theory to a file.

**scattering_factors** *(material, density)*

Returns the scattering factors associated with the material given the range of wavelengths/energies used in the probe.

**shared_beam** *(intensity=1, background=0, back_absorption=1, theta_offset=0)*

Share beam parameters across all segments.

New parameters are created for intensity, background, theta_offset and back_absorption and assigned to the all segments. These can be replaced in an individual segment if that parameter is independent.

**simulate_data** *(theory, noise=2)*

Set the data for the probe to R, adding random noise dR.

If noise is None, then use the uncertainty in the probe.

As a hack, if noise<0, use the probe uncertainty but don’t add noise to the data. Don’t depend on this behavior.

**stitch** *(same_Q=0.001, same_dQ=0.001)*

Stitch together multiple datasets into a single dataset.

Points within tol of each other and with the same resolution are combined by interpolating them to a common Q value then averaged using Gaussian error propagation.

**Returns** probe | Probe Combined data set.

**Algorithm**

To interpolate a set of points to a common value, first find the common Q value:

\[
\hat{Q} = \frac{\sum Q_k}{n}
\]

Then for each dataset k, find the interval \([i, i+1]\) containing the value Q, and use it to compute interpolated value for R:

\[
w = (\hat{Q} - Q_i)/(Q_{i+1} - Q_i)
\]

\[
\hat{R} = w R_{i+1} + (1 - w) R_i
\]

\[
\hat{\sigma}_R = \sqrt{w^2 \sigma^2_{R_i} + (1 - w)^2 \sigma^2_{R_{i+1}}} / n
\]

Average the resulting R using Gaussian error propagation:

\[
\hat{R} = \frac{\sum \hat{R}_k}{n}
\]

\[
\hat{\sigma}_R = \sqrt{\sum \hat{\sigma}^2_{R_k}} / n
\]

**subsample** *(dQ)*

Select points at most every dQ.

Use this to speed up computation early in the fitting process.

This changes the data object, and is not reversible.

The current algorithm is not picking the “best” Q value, just the nearest, so if you have nearby Q points with different quality statistics (as happens in overlapped regions from spallation source measurements at different angles), then it may choose badly. Simple solutions based on the smallest relative error dR/R will be biased toward peaks, and smallest absolute error dR will be biased toward valleys.

**unique_L**
view = ‘fresnel’

write_data(filename, columns=('Q', 'R', 'dR'), header=None)
    Save the data to a file.

    header is a string with trailing n containing the file header. columns is a list of column names from Q, dQ, R, dR, L, dL, T, dT.

    The default is to write Q, R, dR data.

class refl1d.probe.QProbe(Q, dQ, data=None, name=None, filename=None, intensity=1, background=0, back_absorption=1, back_reflectivity=False)

    Bases: refl1d.probe.Probe

    A pure Q, R probe

    This probe with no possibility of tricks such as looking up the scattering length density based on wavelength, or adjusting for alignment errors.

    Aguide = 270

Q

alignment_uncertainty(w, l, d=0)
    Compute alignment uncertainty.

    Parameters:

    w [float | degrees] Rocking curve full width at half max.
    l [float | counts] Rocking curve integrated intensity.
    d = 0: float | degrees Motor step size

    Returns:

    dtheta [float | degrees] uncertainty in alignment angle

apply_beam(calc_Q, calc_R, resolution=True, interpolation=0)
    Apply factors such as beam intensity, background, backabsorption, resolution to the data.

calc_Q

critical_edge(substrate=None, surface=None, n=51, delta=0.25)
    Oversample points near the critical edge.

    The critical edge is defined by the difference in scattering potential for the substrate and surface materials, or the reverse if back_reflectivity is true.

    n is the number of Q points to compute near the critical edge.

    delta is the relative uncertainty in the material density, which defines the range of values which are calculated.

    The n points $Q_i$ are evenly distributed around the critical edge in $Q_c \pm \delta Q_c$ by varying angle $\theta$ for a fixed wavelength $\lambda$, the average of all wavelengths in the probe.

    Specifically:

    $$Q_c^2 = 16\pi(\rho - \rho_{\text{incident}})$$
    $$Q_i = Q_c - \delta_i Q_c(i - (n - 1)/2) \text{ for } i \in 0 \ldots n - 1$$
    $$\lambda_i = <\lambda>$$
    $$\theta_i = \sin^{-1}(Q_i\lambda_i/4\pi)$$

    If $Q_c$ is imaginary, then $-|Q_c|$ is used instead, so this routine can be used for reflectivity signals which scan from back reflectivity to front reflectivity. For completeness, the angle $\theta = 0$ is added as well.
fresnel (substrate=None, surface=None)

    Returns a Fresnel reflectivity calculator given the surface and and substrate. The calculated reflectivity includes The Fresnel reflectivity for the probe reflecting from a block of material with the given substrate.

    Returns \( F = R(\text{probe.Q}) \), where \( R \) is magnitude squared reflectivity.

label (prefix=None, gloss='', suffix='')

    Convert data from log to linear.

    Older reflectometry reduction code stored reflectivity in log base 10 format. Call \text{probe.log10_to_linear()} after loading this data to convert it to linear for subsequent display and fitting.

oversample (n=20, seed=1)

    Generate an over-sampling of Q to avoid aliasing effects.

    Oversampling is needed for thick layers, in which the underlying reflectivity oscillates so rapidly in Q that a single measurement has contributions from multiple Kissig fringes.

    Sampling will be done using a pseudo-random generator so that accidental structure in the function does not contribute to the aliasing. The generator will usually be initialized with a fixed \text{seed} so that the point selection will not change from run to run, but a \text{seed} of \text{None} will choose a different set of points each time \text{oversample} is called.

    The value \( n \) is the number of points that should contribute to each Q value when computing the resolution. These will be distributed about the nominal measurement value, but varying in both angle and energy according to the resolution function. This will yield more points near the measurement and fewer farther away. The measurement point itself will not be used to avoid accidental bias from uniform Q steps. Depending on the problem, a value of \( n \) between 20 and 100 should lead to stable values for the convolved reflectivity.

parameters ()

plot (view=None, **kwargs)

    Plot theory against data.

    Need substrate/surface for Fresnel-normalized reflectivity

plot_Q4 (**kwargs)

    Plot the Q**4 reflectivity associated with the probe.

    Note that Q**4 reflectivity has the intensity and background applied so that hydrogenated samples display more cleanly. The formula to reproduce the graph is:

\[
    R' = R'/(100 * Q)^{-4} I + B) \Delta R' = \Delta R'/(100 * Q)^{-4} I + B)
\]

    where \( B \) is the background.

plot_fft (theory=None, suffix='', label=None, substrate=None, surface=None, **kwargs)

    FFT analysis of reflectivity signal.

plot_fresnel (substrate=None, surface=None, **kwargs)

    Plot the Fresnel-normalized reflectivity associated with the probe.

    Note that the Fresnel reflectivity has the intensity and background applied before normalizing so that hydrogenated samples display more cleanly. The formula to reproduce the graph is:

\[
    R' = R/(F(Q) I + B) \Delta R' = \Delta R/(F(Q) I + B)
\]

    where \( I \) is the intensity and \( B \) is the background.

plot_linear (**kwargs)

    Plot the data associated with probe.
plot_log(**kwargs)
  Plot the data associated with probe.

plot_logfresnel(*args, **kw)
  Plot the log Fresnel-normalized reflectivity associated with the probe.

plot_residuals(theory=None, suffix='', label=None, plot_shift=None, **kwargs)

plot_resolution(suffix='', label=None, **kwargs)

plot_shift = 0
polarized = False
residuals_shift = 0
resolution_guard()
  Make sure each measured \( Q \) point has at least 5 calculated \( Q \) points contributing to it in the range \([-3\Delta Q, 3\Delta Q]\).

Not Implemented

restore_data()
  Restore the original data.

resynth_data()
  Generate new data according to the model \( R \sim N(R_0, dR) \).
  The resynthesis step is a precursor to refitting the data, as is required for certain types of monte carlo error analysis.

save(filename, theory, substrate=None, surface=None)
  Save the data and theory to a file.

scattering_factors(material, density)
  Returns the scattering factors associated with the material given the range of wavelengths/energies used in the probe.

simulate_data(theory, noise=None)
  Set the data for the probe to \( R \), adding random noise \( dR \).
  If noise is None, then use the uncertainty in the probe.
  As a hack, if noise<0, use the probe uncertainty but don’t add noise to the data. Don’t depend on this behavior.

subsample(dQ)
  Select points at most every \( dQ \).
  Use this to speed up computation early in the fitting process.
  This changes the data object, and is not reversible.
  The current algorithm is not picking the “best” \( Q \) value, just the nearest, so if you have nearby \( Q \) points with different quality statistics (as happens in overlapped regions from spallation source measurements at different angles), then it may choose badly. Simple solutions based on the smallest relative error \( dR/R \) will be biased toward peaks, and smallest absolute error \( dR \) will be biased toward valleys.

view = ‘fresnel’

write_data(filename, columns=(‘Q’, ‘R’, ‘dR’), header=None)
  Save the data to a file.

header is a string with trailing n containing the file header. columns is a list of column names from Q, dQ, R, dR, L, dL, T, dT.
The default is to write Q, R, dR data.

```python
refl1d.probe.Qmeasurement_union(xs)
```

Determine the unique Q, dQ across all datasets.

```python
class refl1d.probe.XrayProbe(T=None, dT=0, L=None, dL=0, data=None, intensity=1, background=0, back_absorption=1, theta_offset=0, back_reflectivity=False, name=None, filename=None)
```

Bases: `refl1d.probe.Probe`

X-Ray probe.

By providing a scattering factor calculator for X-ray scattering, model components can be defined by mass density and chemical composition.

```python
Aguide = 270
```

Q

```python
alignment_uncertainty(w, I, d=0)
```

Compute alignment uncertainty.

**Parameters:**

- `w` [float | degrees] Rocking curve full width at half max.
- `I` [float | counts] Rocking curve integrated intensity.
- `d = 0`: float | degrees Motor step size

**Returns:**

- `dtheta` [float | degrees] uncertainty in alignment angle

```python
apply_beam(calc_Q, calc_R, resolution=True, interpolation=0)
```

Apply factors such as beam intensity, background, backabsorption, resolution to the data.

```python
calc_Q
```

```python
critical_edge(substrate=None, surface=None, n=51, delta=0.25)
```

Oversample points near the critical edge.

The critical edge is defined by the difference in scattering potential for the substrate and surface materials, or the reverse if `back_reflectivity` is true.

- `n` is the number of Q points to compute near the critical edge.
- `delta` is the relative uncertainty in the material density, which defines the range of values which are calculated.

The n points $Q_i$ are evenly distributed around the critical edge in $Q_c \pm \delta Q_c$ by varying angle $\theta$ for a fixed wavelength $<\lambda>$, the average of all wavelengths in the probe.

Specifically:

$$Q^2_c = 16\pi(\rho - \rho_{\text{incident}})$$

$$Q_i = Q_c - \delta Q_c(i - (n - 1)/2) \quad \text{for } i \in 0 \ldots n - 1$$

$$\lambda_i = <\lambda>$$

$$\theta_i = \sin^{-1}(Q_i\lambda_i/4\pi)$$

If $Q_c$ is imaginary, then $-|Q_c|$ is used instead, so this routine can be used for reflectivity signals which scan from back reflectivity to front reflectivity. For completeness, the angle $\theta = 0$ is added as well.
fresnel (substrate=None, surface=None)

Returns a Fresnel reflectivity calculator given the surface and and substrate. The calculated reflectivity includes The Fresnel reflectivity for the probe reflecting from a block of material with the given substrate.

Returns \( F = R(probe.Q) \), where \( R \) is magnitude squared reflectivity.

label (prefix=None, gloss='', suffix='')

log10_to_linear ()

Convert data from log to linear.

Older reflectometry reduction code stored reflectivity in log base 10 format. Call probe.log10_to_linear() after loading this data to convert it to linear for subsequent display and fitting.

oversample (n=20, seed=1)

Generate an over-sampling of Q to avoid aliasing effects.

Oversampling is needed for thick layers, in which the underlying reflectivity oscillates so rapidly in Q that a single measurement has contributions from multiple Kissig fringes.

Sampling will be done using a pseudo-random generator so that accidental structure in the function does not contribute to the aliasing. The generator will usually be initialized with a fixed \( seed \) so that the point selection will not change from run to run, but a \( seed \) of None will choose a different set of points each time oversample is called.

The value \( n \) is the number of points that should contribute to each \( Q \) value when computing the resolution. These will be distributed about the nominal measurement value, but varying in both angle and energy according to the resolution function. This will yield more points near the measurement and fewer farther away. The measurement point itself will not be used to avoid accidental bias from uniform Q steps. Depending on the problem, a value of \( n \) between 20 and 100 should lead to stable values for the convolved reflectivity.

parameters ()

plot (view=None, **kwargs)

Plot theory against data.

Need substrate/surface for Fresnel-normalized reflectivity

plot_Q4 (**kwargs)

Plot the \( Q^4 \) reflectivity associated with the probe.

Note that \( Q^4 \) reflectivity has the intensity and background applied so that hydrogenated samples display more cleanly. The formula to reproduce the graph is:

\[
R' = R/((100 * Q)^{-4} I + B) \Delta R' = \Delta R/((100 * Q)^{-4} I + B)
\]

where \( B \) is the background.

plot_fft (theory=None, suffix='', label=None, substrate=None, surface=None, **kwargs)

FFT analysis of reflectivity signal.

plot_fresnel (substrate=None, surface=None, **kwargs)

Plot the Fresnel-normalized reflectivity associated with the probe.

Note that the Fresnel reflectivity has the intensity and background applied before normalizing so that hydrogenated samples display more cleanly. The formula to reproduce the graph is:

\[
R' = R/(F(Q)I + B) \Delta R' = \Delta R/(F(Q)I + B)
\]

where \( I \) is the intensity and \( B \) is the background.

plot_linear (**kwargs)

Plot the data associated with probe.
plot_log(**kwargs)
    Plot the data associated with probe.

plot_logfresnel(*args, **kw)
    Plot the log Fresnel-normalized reflectivity associated with the probe.

plot_residuals(theory=None, suffix='', label=None, plot_shift=None, **kwargs)

plot_resolution(suffix='', label=None, **kwargs)

plot_shift = 0
polarized = False
radiation = 'xray'
residuals_shift = 0
resolution_guard()
    Make sure each measured $Q$ point has at least 5 calculated $Q$ points contributing to it in the range $[-3\Delta Q, 3\Delta Q]$.
    Not Implemented
restore_data()
    Restore the original data.
resynth_data()
    Generate new data according to the model $R \sim N(R_0, dR)$.
    The resynthesis step is a precursor to refitting the data, as is required for certain types of monte carlo error analysis.
save(filename, theory, substrate=None, surface=None)
    Save the data and theory to a file.
scattering_factors(material, density)
    Returns the scattering factors associated with the material given the range of wavelengths/energies used in the probe.
simulate_data(theory, noise=None)
    Set the data for the probe to $R$, adding random noise $dR$.
    If noise is None, then use the uncertainty in the probe.
    As a hack, if noise<0, use the probe uncertainty but don’t add noise to the data. Don’t depend on this behavior.
subsample(dQ)
    Select points at most every dQ.
    Use this to speed up computation early in the fitting process.
    This changes the data object, and is not reversible.
    The current algorithm is not picking the “best” $Q$ value, just the nearest, so if you have nearby $Q$ points with different quality statistics (as happens in overlapped regions from spallation source measurements at different angles), then it may choose badly. Simple solutions based on the smallest relative error $dR/R$ will be biased toward peaks, and smallest absolute error $dR$ will be biased toward valleys.
view = 'fresnel'
write_data(filename, columns=('Q', 'R', 'dR'), header=None)
    Save the data to a file.
Header is a string with trailing n containing the file header. Columns is a list of column names from Q, dQ, R, dR, L, dL, T, dT.

The default is to write Q, R, dR data.

refl1d.probe.load4(filename, keysep=':', sep=None, comment='#', name=None, intensity=1, background=0, back_absorption=1, back_reflectivity=False, Aguide=270, H=0, theta_offset=0, sample_broadening=None, L=None, dL=None, T=None, dT=None, FWHM=False)

Load in four column data Q, R, dR, dQ.

The file is loaded with bumps.data.parse_multi. keysep defaults to ':' so that header data looks like JSON key: value pairs. sep is None so that the data uses white-space separated columns. comment is the standard '#' comment character, used for "# key: value" lines, for commenting out data lines using "#number number number number", and for adding comments after individual data lines. The parser isn’t very sophisticated, so be nice.

Intensity is the overall beam intensity, background is the overall background level, and back_absorption is the relative intensity of data measured at negative Q compared to positive Q data. These can be values or a bumps Parameter objects.

back_reflectivity is True if reflectivity was measured through the substrate. This allows you to arrange the model from substrate to surface regardless of whether you are measuring through the substrate or reflecting off the surface.

theta_offset indicates sample alignment. In order to use theta offset you need to be able to convert from Q to wavelength and angle by providing values for the wavelength or the angle, and the associated resolution.

For monochromatic sources you can supply L, dLoL when you call load4, or you can store it in the header of the file:

```plaintext
# wavelength: 4.75  # Ang
# wavelength_resolution: 0.02  # Ang (1-sigma)
```

For time of flight sources, angle is fixed and wavelength is varying, so you can supply T, dT in degrees when you call load4, or you can store it in the header of the file:

```plaintext
# angle: 2  # degrees
# angular_resolution: 0.2  # degrees (1-sigma)
```

If both angle and wavelength are varying in the data, you can specify a separate value for each point, such the following:

```plaintext
# wavelength: [1, 1.2, 1.5, 2.0, ...]
# wavelength_resolution: [0.02, 0.02, 0.02, ...]
```

Sample broadening in degrees (1-σ) adds to the angular_resolution.

Aguide and H are parameters for polarized beam measurements indicating the magnitude and direction of the applied field.

Polarized data is represented using a multi-section data file, with blank lines separating each section. Each section must have a polarization keyword, with value “++”, “+-”, “-+” or “--”.

FWHM is True if dQ, dT, dL are given as FWHM rather than 1-σ. dR is always 1-σ.

refl1d.probe.make_probe(**kw)
Return a reflectometry measurement object of the given resolution.

refl1d.probe.measurement_union(xs)
Determine the unique (T, dT, L, dL) across all datasets.

4.19. probe - Instrument probe
refl1d.probe.spin_asymmetry \((Q_p, R_p, dR_p, Q_m, R_m, dR_m)\)
Compute spin asymmetry for \(R^{++}, R^{-}\).

**Parameters:**

- \(Q_p, R_p, dR_p\) [vector] Measured ++ cross section and uncertainty.
- \(Q_m, R_m, dR_m\) [vector] Measured – cross section and uncertainty.

If \(dR_p, dR_m\) are None then the returned uncertainty will also be None.

**Returns:**

- \(Q, SA, dSA\) [vector] Computed spin asymmetry and uncertainty.

**Algorithm:**

Spin asymmetry, \(S_A\), is:

\[
S_A = \frac{R^{++} - R^{--}}{R^{++} + R^{--}}
\]

Uncertainty \(\Delta S_A\) follows from propagation of error:

\[
\Delta S_A^2 = \frac{4(R^{++}_1 \Delta R^{2} + R^{2} \Delta R^{++})}{(R^{++} + R^{--})^4}
\]

### 4.20 profile - Model profile

<table>
<thead>
<tr>
<th>Microslabs</th>
<th>Manage the micro slab representation of a model.</th>
</tr>
</thead>
<tbody>
<tr>
<td>blend</td>
<td>blend function</td>
</tr>
<tr>
<td>build_mag_profile</td>
<td>Convert magnetic segments to a smooth profile.</td>
</tr>
<tr>
<td>build_profile</td>
<td>Convert a step profile to a smooth profile.</td>
</tr>
<tr>
<td>compute_limited_sigma</td>
<td></td>
</tr>
</tbody>
</table>

Scattering length density profile.

In order to render a reflectometry model, the theory function calculator renders each layer in the model for each energy in the probe. For slab layers this is easy: just accumulate the slabs, with the 1-\(\sigma\) Gaussian interface width between the slabs. For freeform or functional layers, this is more complicated. The rendering needs to chop each layer into microslabs and evaluate the profile at each of these slabs.

#### 4.20.1 Example

This example sets up a model which uses tanh to transition from silicon to gold in 20 Å with 2 Å steps.

First define the profile, and put in the substrate:

```python
>>> S = Microslabs(nprobe=1,dz=2)
>>> S.clear()
>>> S.append(w=0,rho=2.07)
```

Next add the interface. This uses `microslabs()` to select the points at which the interface is evaluated, much like you would do when defining your own special layer type. Note that the points \(P_z\) are in the center of the micro slabs. The width of the final slab may be different. You do not need to use fixed width microslabs if you can more efficiently
represent the profile with a smaller number of variable width slabs, but `contract_profile()` serves the same purpose with less work on your part.

```python
>>> from numpy import tanh
>>> Pw,Pz = S.microslabs(20)
>>> print("widths = \$s ...\%" \ " \ .join("%g\%v for v in Pw[:5]))"
widths = 2 2 2 2 2 ...
```

```python
>>> print("centers = \$s ...\%" \ " \ .join("%g\%v for v in Pz[:5)))"
centers = 1 3 5 7 9 ...
```

```python
>>> rho = (1-tanh((Pz-10)/5))/2*(2.07-4.5)+4.5
>>> S.extend(w=Pw, rho=[rho])
```

Finally, add the incident medium and see the results. Note that `rho` is a matrix, with one column for each incident energy. We are only using one energy so we only show the first column.

```python
>>> S.append(w=0, rho=4.5)
>>> print("width = \$s ...\%" \ " \ .join("%g\%v for v in S.w[:5]))"
width = 0 2 2 2 2 ...
```

```python
>>> print("rho = \$s ...\%" \ " \ .join("%.2f\%v for v in S.rho[0,:5]))"
rho = 2.07 2.13 2.21 2.36 2.63 ...
```

Since `irho` and `sigma` were not specified, they will be zero.

```python
>>> print("sigma = \$s ...\%" \ " \ .join("%g\%v for v in S.sigma[:5]))"
sigma = 0 0 0 0 0 ...
```

```python
>>> print("irho = \$s ...\%" \ " \ .join("%g\%v for v in S.irho[0,:5]))"
irho = 0 0 0 0 0 ...
```

### class refl1d.profile.Microslabs (nprobe, dz=1)

Bases: object

Manage the micro slab representation of a model.

In order to compute reflectivity, we need a series of slabs with thickness, roughness and scattering potential for each slab. Because scattering potentials are probe dependent we store an array of potentials for each probe value.

Some slab models use non-uniform layers, and so need the additional parameter of `dz` for the step size within the layer.

The space for the slabs is saved even after reset, in preparation for a new set of slabs from different fitting parameters.

- **add_magnetism** (anchor, w, rhoM=0, thetaM=270.0, sigma=0)
  - Add magnetic layers.

- **append** (w=0, sigma=0, rho=0, irho=0)
  - Extend the micro slab model with a single layer.

- **clear**
  - Reset the slab model so that none are present.

- **extend** (w=0, sigma=0, rho=0, irho=0)
  - Extend the micro slab model with the given layers.

- **finalize** (step_interfaces, dA, roughness_limit)
  - Rendering complete.

Call this method after the microslab model has been constructed, so any post-rendering processes can be completed.
In addition to clearing any width from the substrate and the surface surround, this will align magnetic and nuclear slabs, convert interfaces to step interfaces if desired, and merge slabs with similar scattering potentials to reduce computation time.

*step_interfaces* is True if interfaces should be rendered using slabs.

dA is the tolerance to use when deciding if similar layers can be merged.

*roughness_limit* is the maximum

**interface** (I)

Interfaces act to smear the microslabs after the fact. This allows more flexibility than trying to compute the effects of roughness on non-flat layers.

*irho*

Absorption (10^-6 number density)

*ismagnetic*

**magnetic_profile** ()

Return a profile representation of the magnetic microslab structure.

**microslabs** *(thickness=0)*

Return a set of microslabs for a layer of the given thickness.

The step size slabs.dz was defined when the Microslabs object was created.

This is a convenience function. Layer definitions can choose their own slices so long as the step size is approximately slabs.dz in the varying region.

**Parameters**

*thickness* [float | A] Layer thickness

**Returns**

*widths*: vector | A Microslab widths

*centers*: vector | A Microslab centers

**repeat** *(start=0, count=1, interface=0)*

Extend the model so that there are count versions of the slabs from start to the final slab.

This is equivalent to L.extend(L[start:]*(count-1)) for list L.

*rho*

Scattering length density (10^-6 number density)

**sigma**

rms roughness (Å)

**smooth_profile** *(dz=1)*

Return a smooth profile representation of the microslab structure

Nevot-Croce roughness is approximately represented, though the calculation is incorrect for layers with large roughness compared to the thickness.

The returned profile has uniform step size dz.

**step_profile** ()

Return a step profile representation of the microslab structure.

Nevot-Croce interfaces are not represented.

**surface_sigma**

sigma above the surface (which is not part of sigma)
thickness()  
Total thickness of the profile.

Note that thickness includes the thickness of the substrate and surface layers. Normally these will be zero, but the contract profile operation may result in large values for either.

w  
Thickness (Å)

refl1d.profile.blend(z, rough)  
blend function  
Given a Gaussian roughness value, compute the portion of the neighboring profile you expect to find in the current profile at depth z.

refl1d.profile.build_mag_profile(z, d, v, blends)  
Convert magnetic segments to a smooth profile.

refl1d.profile.build_profile(z, thickness, roughness, value)  
Convert a step profile to a smooth profile.

z calculation points thickness thickness of the layers (first and last values ignored) roughness roughness of the interfaces (one less than d) value profile being computed max_rough limit the roughness to a fraction of the layer thickness

refl1d.profile.compute_limited_sigma(thickness, roughness, limit)

4.21 reflectivity - Reflectivity

| reflectivity          | Calculate reflectivity $|r(k_z)|^2$ from slab model. |
|-----------------------|--------------------------|
| reflectivity_amplitude| Calculate reflectivity amplitude $r(k_z)$ from slab model. |
| magnetic_reflectivity | Magnetic reflectivity for slab models. |
| magnetic_amplitude    | Returns the complex magnetic reflectivity waveform. |
| unpolarized_magnetic  | Returns the average of magnetic reflectivity for all cross-sections. |
| convolve              | Apply x-dependent gaussian resolution to the theory. |

Basic reflectometry calculations

Slab model reflectivity calculator with optional absorption and roughness. The function reflectivity_amplitude returns the complex waveform. Slab model with supporting magnetic scattering. The function magnetic_reflectivity returns the complex reflection for the four spin polarization cross sections $[++, +-, -+, –]$. The function unpolarized_magnetic returns the expected magnitude for a measurement of the magnetic scattering using an unpolarized beam.

refl1d.reflectivity.reflectivity(*args, **kw)  
Calculate reflectivity $|r(k_z)|^2$ from slab model.

:Parameters:

- **depth [float[N] | Å]** Thickness of the individual layers (incident and substrate depths are ignored)

- **sigma [float OR float[N-1] | Å]** Interface roughness between the current layer and the next. The final layer is ignored. This may be a scalar for fixed roughness on every layer, or None if there is no roughness.

- **rho, irho [float[N] OR float[N,K] | 10^{-6}Å^{-2}]** Real and imaginary scattering length density. Use multiple columns when you have kz-dependent scattering length densities, and set rho_offset to select the appropriate one. Data should be stored in column order.

- **kz [float[M] | Å^{-1}]** Points at which to evaluate the reflectivity

4.21. reflectivity - Reflectivity 127
rho_index  [integer[M]] rho and irho columns to use for the various kz.

Returns

\[ R | \text{float[M]} \] Reflectivity magnitude.

This function does not compute any instrument resolution corrections.

```
refl1d.reflectivity.reflectivity_amplitude(kz=None, depth=None, rho=None, irho=0, sigma=0, rho_index=None)
```

Calculate reflectivity amplitude \( r(k_z) \) from slab model.

:Parameters:

- `depth`  [float[N] | Å] Thickness of the individual layers (incident and substrate depths are ignored)
- `sigma` = 0 [float OR float[N-1] | Å] Interface roughness between the current layer and the next. The final layer is ignored. This may be a scalar for fixed roughness on every layer, or None if there is no roughness.
- `rho, irho = 0: float[N] OR float[N,K] | 10^{-6} \text{Å}^{-2}` Real and imaginary scattering length density. Use multiple columns when you have kz-dependent scattering length densities, and set rho_index to select amongst them. Data should be stored in column order.
- `kz`  [float[M] | Å^{-1}] Points at which to evaluate the reflectivity
- `rho_index = 0`  [integer[M]] rho and irho columns to use for the various kz.

Returns

\[ r | \text{complex[M]} \] Complex reflectivity waveform.

This function does not compute any instrument resolution corrections.

```
refl1d.reflectivity.magnetic_reflectivity(*args, **kw)
```

Magnetic reflectivity for slab models.

Returns the expected values for the four polarization cross sections (++,+-,-+,–). Return reflectivity \( R^2 \) from slab model with sharp interfaces. returns reflectivities.

The parameters are as follows:

- `kz` (Å^{-1}) points at which to evaluate the reflectivity
- `depth` (Å) thickness of the individual layers (incident and substrate depths are ignored)
- `rho` (microNb) Scattering length density.
- `mu` (microNb) absorption. Defaults to 0.
- `wavelength` (Å) Incident wavelength (only affects absorption). May be a vector. Defaults to 1.
- `rho_m` (microNb) Magnetic scattering length density correction.
- `theta_m` (degrees) Angle of the magnetism within the layer.
- `Aguide` (degrees) Angle of the guide field; -90 is the usual case

This function does not compute any instrument resolution corrections or interface diffusion

Use magnetic_amplitude to return the complex waveform.

```
refl1d.reflectivity.magnetic_amplitude(kz, depth, rho, irho=0, rhoM=0, thetaM=0, sigma=0, Aguide=-90, H=0, rho_index=None, rotate_M=True)
```

Returns the complex magnetic reflectivity waveform.
Refl1D: Neutron and X-Ray Reflectivity Analysis, Release 0.7.8

See `magnetic_reflectivity` for details.

```python
ref1d.reflectivity.unpolarized_magnetic(*args, **kw)
```

Returns the average of magnetic reflectivity for all cross-sections.

See `magnetic_reflectivity` for details.

```python
ref1d.reflectivity.convolve(xi, yi, x, dx)
```

Apply x-dependent gaussian resolution to the theory.

Returns convolution `y[k]` of width `dx[k]` at points `x[k]`.

The theory function is a piece-wise linear spline which does not need to be uniformly sampled. The theory calculation points `xi` should be dense enough to capture the “wiggle” in the theory function, and should extend beyond the ends of the data measurement points `x`. Convolution at the tails is truncated and normalized to area of overlap between the resolution function in case the theory does not extend far enough.

### 4.22 reflmodule - Low level reflectivity calculations

<table>
<thead>
<tr>
<th>Convolve Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>convolve(xi, yi, x, dx)</code></td>
<td>Compute convolution of width <code>dx[k]</code> at points <code>x[k]</code>.</td>
</tr>
<tr>
<td><code>convolve_sampled(xi, yi, x, yp, x, dx)</code></td>
<td>Compute convolution with sampled distribution of width <code>dx[k]</code> at points <code>x[k]</code>.</td>
</tr>
<tr>
<td><code>rebin2d_float32(xi, yi, x, xo, Io)</code></td>
<td>2-D rebin from <code>xi</code> to <code>xo</code></td>
</tr>
<tr>
<td><code>rebin2d_float64(xi, yi, x, xo, Io)</code></td>
<td>2-D rebin from <code>xi</code> to <code>xo</code></td>
</tr>
<tr>
<td><code>rebin2d_uint16(xi, yi, x, xo, Io)</code></td>
<td>2-D rebin from <code>xi</code> to <code>xo</code></td>
</tr>
<tr>
<td><code>rebin2d_uint32(xi, yi, x, xo, Io)</code></td>
<td>2-D rebin from <code>xi</code> to <code>xo</code></td>
</tr>
<tr>
<td><code>rebin2d_uint64(xi, yi, x, xo, Io)</code></td>
<td>2-D rebin from <code>xi</code> to <code>xo</code></td>
</tr>
<tr>
<td><code>rebin2d_uint8(xi, yi, x, xo, Io)</code></td>
<td>2-D rebin from <code>xi</code> to <code>xo</code></td>
</tr>
<tr>
<td><code>rebin_float32(xi, yi, x, xo, Io)</code></td>
<td>2-D rebin from <code>xi</code> to <code>xo</code></td>
</tr>
</tbody>
</table>

Reflectometry C Library

```python
ref1d.reflmodule.convolve() 
    convolve(xi, yi, x, dx): compute convolution of width `dx[k]` at points `x[k]`, returned in `y[k]`

ref1d.reflmodule.convolve_sampled() 
    convolve_sampled(xi, yi, x, yp, x, dx): compute convolution with sampled distribution of width `dx[k]` at points `x[k]`, returned in `y[k]`

ref1d.reflmodule.rebin2d_float32 (xi, yi, x, xo, Io): 2-D rebin from `xi` to `xo` 
ref1d.reflmodule.rebin2d_float64 (xi, yi, x, xo, Io): 2-D rebin from `xi` to `xo` 
ref1d.reflmodule.rebin2d_uint16 (xi, yi, x, xo, Io): 2-D rebin from `xi` to `xo` 
ref1d.reflmodule.rebin2d_uint32 (xi, yi, x, xo, Io): 2-D rebin from `xi` to `xo` 
ref1d.reflmodule.rebin2d_uint64 (xi, yi, x, xo, Io): 2-D rebin from `xi` to `xo` 
ref1d.reflmodule.rebin2d_uint8 (xi, yi, x, xo, Io): 2-D rebin from `xi` to `xo` 
ref1d.reflmodule.rebin_float32 () 
    rebin_float32(xi, yi, x, xo, Io): rebin from bin edges `xi` to bin edges `xo`
refl1d.reflmodule.rebin_float64()  
rebin_float64(xi,li,xi,lo): rebin from bin edges xi to bin edges xo

refl1d.reflmodule.rebin_uint16()  
rebin_uint16(xi,li,xi,lo): rebin from bin edges xi to bin edges xo

refl1d.reflmodule.rebin_uint32()  
rebin_uint32(xi,li,xi,lo): rebin from bin edges xi to bin edges xo

refl1d.reflmodule.rebin_uint64()  
rebin_uint64(xi,li,xi,lo): rebin from bin edges xi to bin edges xo

refl1d.reflmodule.rebin_uint8()  
rebin_uint8(xi,li,xi,lo): rebin from bin edges xi to bin edges xo

4.23 resolution - Resolution

\[\text{FWHM2sigma} \]
Compute angle from \( Q \) and wavelength.

\[\text{QL2T} \quad (Q=None, L=None)\]
Compute \( Q \) from angle and wavelength.

\[\theta = \sin^{-1}(|Q|\lambda/4\pi)\]

Returns \( \theta^\circ \).

\[\text{QT2L} \quad (Q=None, T=None)\]
Compute wavelength from \( Q \) and angle.

\[\lambda = 4\pi \sin(\theta)/Q\]

Returns \( \lambda \text{Å} \).

\[\text{TL2Q} \quad (T=None, L=None)\]
Compute \( Q \) from angle and wavelength.

\[Q = 4\pi \sin(\theta)/\lambda\]

Returns \( Q \text{ Å}^{-1} \)
refl1d.resolution.TOF2L (d_moderator, TOF)
Convert neutron time-of-flight to wavelength.

\[ \lambda = \left( \frac{t}{d} \right) \left( \frac{h}{n_m} \right) \]

where:
- \( \lambda \) is wavelength in Å
- \( t \) is time-of-flight in \( \mu s \)
- \( h \) is Planck’s constant in erg seconds
- \( n_m \) is the neutron mass in g

refl1d.resolution.binedges (L)
Construct bin edges \( E \) from bin centers \( L \).

Assuming fixed \( \omega = \Delta \lambda / \lambda \) in the bins, the edges will be spaced logarithmically at:

\[ E_0 = \min \lambda \]
\[ E_{i+1} = E_i + \omega E_i = E_i (1 + \omega) \]

with centers \( L \) half way between the edges:

\[ L_i = \frac{(E_i + E_{i+1})/2}{(E_i + E_{i+1})}/2 = E_i (2 + \omega)/2 \]

Solving for \( E_i \), we can recover the edges from the centers:

\[ E_i = L_i \frac{2}{2 + \omega} \]

The final edge, \( E_{n+1} \), does not have a corresponding center \( L_{n+1} \) so we must determine it from the previous edge \( E_n \):

\[ E_{n+1} = L_n \frac{2}{2 + \omega} (1 + \omega) \]

The fixed \( \omega \) can be retrieved from the ratio of any pair of bin centers using:

\[ \frac{L_{i+1}}{L_i} = \frac{(E_{i+2} + E_{i+1})/2}{(E_{i+1} + E_i)/2} = \frac{(E_{i+1}(1 + \omega) + E_{i+1})}{(E_i(1 + \omega) + E_i)} = \frac{E_{i+1}}{E_i} = \frac{E_i(1 + \omega)}{E_i} = 1 + \omega \]

refl1d.resolution.bins (low, high, dLoL)
Return bin centers from low to high preserving a fixed resolution. 

\( low, high \) are the minimum and maximum wavelength. \( dLoL \) is the desired resolution FWHM \( \Delta \lambda / \lambda \) for the bins.

refl1d.resolution.binwidths (L)
Determine the wavelength dispersion from bin centers \( L \).

The wavelength dispersion \( \Delta \lambda \) is just the difference between consecutive bin edges, so:

\[ \Delta L_i = E_{i+1} - E_i = (1 + \omega)E_i - E_i = \omega E_i = \frac{2 \omega}{2 + \omega} L_i \]

where \( E \) and \( \omega \) are as defined in binedges().

refl1d.resolution.dQdL2dT (Q, dQ, L, dL)
Convert a calculated Q resolution and wavelength dispersion to angular divergence.

\( Q, dQ \) \( \text{Å}^{-1} \) and 1-\( \sigma \) \( Q \) resolution \( L, dL \) \( \circ \) angle and FWHM angular divergence

Returns FWHM Deltatheta$
refl1d.resolution.dQdT2dLoL(Q, dQ, T, dT)
Convert a calculated Q resolution and angular divergence to a wavelength dispersion.

Q, dQ Å⁻¹ Q and 1-σ Q resolution T, dT ° angle and FWHM angular divergence
Returns FWHM Δλ/λ

refl1d.resolution.ddTdL2dQ(T=None, dT=None, L=None, dL=None)
Convert wavelength dispersion and angular divergence to Q resolution.

T, dT (degrees) angle and FWHM angular divergence L, dL (Angstroms) wavelength and FWHM wavelength dispersion
Returns 1-σ ΔQ

refl1d.resolution.divergence(T=None, slits=None, distance=None, sample_width=10000000000.0, sample_broadening=0)
Calculate divergence due to slit and sample geometry.

Parameters

T [float OR [float | degrees]] incident angles
slits [float OR (float,float) | mm] s1,s2 slit openings for slit 1 and slit 2
distance [(float,float) | mm] d1,d2 distance from sample to slit 1 and slit 2
sample_width [float | mm] w, width of the sample
sample_broadening [float | degrees FWHM] additional divergence caused by sample

Returns

dT [float OR [float | degrees FWHM]] calculated angular divergence

Algorithm:
The divergence is based on the slit openings and the distance between the slits. For very small samples, where the slit opening is larger than the width of the sample across the beam, the sample itself acts like the second slit.
First find p, the projection of the beam on the sample:

\[ p = w \sin \left( \frac{\pi}{180} \theta \right) \]

Depending on whether p is larger than s2, determine the slit divergence \( \Delta \theta_d \) in radians:

\[ \Delta \theta_d = \begin{cases} \frac{1}{2} \frac{s_1 + s_2}{d_1 + d_2} & \text{if } p \geq s_2 \\ \frac{1}{2} \frac{s_1 + p}{d_1} & \text{if } p < s_2 \end{cases} \]

In addition to the slit divergence, we need to add in any sample broadening \( \Delta \theta_s \) returning the total divergence in degrees:

\[ \Delta \theta = \frac{180}{\pi} \Delta \theta_d + \Delta \theta_s \]

Reversing this equation, the sample broadening contribution can be measured from the full width at half maximum of the rocking curve, B, measured in degrees at a particular angle and slit opening:

\[ \Delta \theta_s = B - \frac{180}{\pi} \Delta \theta_d \]

refl1d.resolution.sigma2FWHM(s)

refl1d.resolution.slit_widths(T=None, slits_at_Tlo=None, Tlo=90, Thi=90, slits_below=None, slits_above=None)
Compute the slit widths for the standard scanning reflectometer fixed-opening-fixed geometry.
Parameters

\[ T \]  \([\text{float} \mid \text{degrees}]\) Specular measurement angles.

\[ T_{lo}, \, T_{hi} \]  \([\text{float} \mid \text{degrees}]\) Start and end of the opening region. The default if \( T_{lo} \) is not specified is to use fixed slits at \( slits\_below \) for all angles.

\[ slits\_below, \, slits\_above \]  \([\text{float OR [float, float]} \mid \text{mm}]\) Slits outside opening region. The default is to use the values of the slits at the ends of the opening region.

\[ slits\_at\_Tlo \]  \([\text{float OR [float, float]} \mid \text{mm}]\) Slits at the start of the opening region.

Returns

\( s1, \, s2 \)  \([\text{float} \mid \text{mm}]\) Slit widths for each theta.

Slits are assumed to be fixed below angle \( T_{lo} \) and above angle \( T_{hi} \), and opening at a constant \( dT/T \) between them.

Slit openings are defined by a tuple \((s1, s2)\) or constant \( s=s1=s2 \). With no \( T_{lo} \), the slits are fixed with widths defined by \( slits\_below \), which defaults to \( slits\_at\_Tlo \). With no \( T_{hi} \), slits are continuously opening above \( T_{lo} \).

**Note:** This function works equally well if angles are measured in radians and/or slits are measured in inches.

### 4.24 snsdata - SNS Data

<table>
<thead>
<tr>
<th>Liquids</th>
<th>Loader for reduced data from the SNS Liquids instrument.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magnetic</td>
<td>Loader for reduced data from the SNS Magnetic instrument.</td>
</tr>
<tr>
<td>QRL_to_data</td>
<td>Convert data to T,L,R</td>
</tr>
<tr>
<td>SNDdata</td>
<td></td>
</tr>
<tr>
<td>TOF_to_data</td>
<td>Convert TOF data to neutron probe.</td>
</tr>
<tr>
<td>boltzmann_feather</td>
<td>Return expected intensity as a function of wavelength given the TOF feather range and the total number of events.</td>
</tr>
<tr>
<td>has_columns</td>
<td></td>
</tr>
<tr>
<td>intensity_from_spline</td>
<td></td>
</tr>
<tr>
<td>load</td>
<td>Return a probe for SNS data.</td>
</tr>
<tr>
<td>parse_sns_file</td>
<td>Parse SNS reduced data, returning header and data.</td>
</tr>
<tr>
<td>write_file</td>
<td>Save probe as SNS reduced file.</td>
</tr>
</tbody>
</table>

SNS data loaders

The following instruments are defined:

Liquids, Magnetic

These are resolution.Pulsed classes tuned with default instrument parameters and loaders for reduced SNS data. See resolution for details.

```python
class refl1d.snsdata.Liquids(**kw)
    Bases: refl1d.snsdata.SNDdata, refl1d.instrument.Pulsed

Loader for reduced data from the SNS Liquids instrument.

T = None
TOF_range = (6000, 60000)
Thi = 90
```
\( T_{\text{lo}} = 90 \)

\[ \text{calc\_dT} \left( T, \text{slits, } **\text{kw} \right) \]

\[ \text{calc\_slits} \left( **\text{kw} \right) \]

Determines slit openings from measurement pattern.

If slits are fixed simply return the same slits for every angle, otherwise use an opening range \([T_{\text{lo}}, T_{\text{hi}}]\) and the value of the slits at the start of the opening to define the slits. Slits below \( T_{\text{lo}} \) and above \( T_{\text{hi}} \) can be specified separately.

\( T \) incident angle \( T_{\text{lo}}, T_{\text{hi}} \) angle range over which slits are opening \( \text{slits\_at\_Tlo} \) openings at the start of the range, or fixed opening \( \text{slits\_below, slits\_above} \) openings below and above the range

Use fixed_slits is available, otherwise use opening slits.

\( d_{\text{LoL}} = 0.02 \)

\( d_{\text{moderator}} = 14.85 \)

\( d_{\text{s1}} = 2086.0 \)

\( d_{\text{s2}} = 230.0 \)

\[ \text{defaults} \left( \right) \]

Return default instrument properties as a printable string.

\[ \text{feather} = \text{array}([[2.02555, 2.29927, 2.57299, 2.87409, 3.22993, 3.58577, 4.07847, 4.5438, 5.11861, 5.7208, 6.37774, ... , 6.59236, 4.68153, 3.05732, 1.91083, 1.24204, 0.955414, 0.573248, 0.477707, 0.382166, 0.191083, 0.286624]]) \]

\[ \text{fixed\_slits} = \text{None} \]

\[ \text{instrument} = \text{‘Liquids’} \]

\[ \text{load} \left( \text{filename, } **\text{kw} \right) \]

\[ \text{magnetic\_probe} \left( \text{Aguide=270, shared\_beam=True, } **\text{kw} \right) \]

Simulate a polarized measurement probe.

Returns a probe with \( Q, \text{ angle}, \text{ wavelength and the associated uncertainties, but not any data.} \)

Guide field angle \( \text{Aguide} \) can be specified, as well as keyword arguments for the geometry of the probe cross sections such as slit settings \( \text{slits} \) and \( T \) to define the angular divergence and \( d_{\text{LoL}} \) to define the wavelength resolution.

\[ \text{probe} \left( **\text{kw} \right) \]

Simulate a measurement probe.

Returns a probe with \( Q, \text{ angle}, \text{ wavelength and the associated uncertainties, but not any data.} \)

You can override instrument parameters using key=value. In particular, slit settings \( \text{slits} \) and \( T \) define the angular divergence and \( d_{\text{LoL}} \) defines the wavelength resolution.

\[ \text{radiation} = \text{‘neutron’} \]

\[ \text{resolution} \left( L, dL, **\text{kw} \right) \]

Return the resolution of the measurement. Needs \( T, L, dL \) specified as keywords.

\[ \text{sample\_broadening} = 0 \]

\[ \text{sample\_width} = 1.0000000000 \]

\[ \text{simulate} \left( \text{sample, uncertainty=1, } **\text{kw} \right) \]

Simulate a run with a particular sample.

**Parameters**

\[ \text{sample} \text{ [Stack] Reflectometry model} \]
\[ T \] [[float] | °] List of angles to be measured, such as [0.15,0.4,1,2].

**slits** [[float] or [(float, float)] | mm] Slit settings for each angle.

**uncertainty** = 1 [float or [float] | %] Incident intensity is set so that the median dR/R is equal to uncertainty, where R is the idealized reflectivity of the sample.

\[ dLoL = 0.02; \text{ float} \] Wavelength resolution

**normalize** = True [boolean] Whether to normalize the intensities

**theta_offset** = 0 [float | °] Sample alignment error

**background** = 0 [float] Background counts per incident neutron (background is assumed to be independent of measurement geometry).

**back_reflectivity** = False [boolean] Whether beam travels through incident medium or through substrate.

**back_absorption** = 1 [float] Absorption factor for beam traveling through substrate. Only needed for back reflectivity measurements.

**slits** = None

**slits_above** = None

**slits_at_Tlo** = None

**slits_below** = None

**wavelength** = (2.0, 15.0)

**class refl1d.snsdata.Magnetic(**

Bases: refl1d.snsdata.SNSData, refl1d.instrument.Pulsed

Loader for reduced data from the SNS Magnetic instrument.

**T** = None

**TOF_range** = (0, inf)

**Thi** = 90

**Tlo** = 90

**calc_dT** (**T, slits**, **kw**)

**calc_slits** (**kw**)

Determines slit openings from measurement pattern.

If slits are fixed simply return the same slits for every angle, otherwise use an opening range \([Tlo, Thi]\) and the value of the slits at the start of the opening to define the slits. Slits below Tlo and above Thi can be specified separately.

\( T \) incident angle \( Tlo, Thi \) angle range over which slits are opening \( \text{slits_at_Tlo} \) openings at the start of the range, or fixed opening \( \text{slits_below, slits_above} \) openings below and above the range

Use fixed_slits is available, otherwise use opening slits.

\[ dLoL = 0.02 \]

\[ d_s1 = 190.5 \]

\[ d_s2 = 35.56 \]

**defaults()**

Return default instrument properties as a printable string.
fixed_slits = None
instrument = 'Magnetic'
load (filename, **kw)
magnetic_probe (Aguide=270, shared_beam=True, **kw)
    Simulate a polarized measurement probe.
    Returns a probe with Q, angle, wavelength and the associated uncertainties, but not any data.
    Guide field angle Aguide can be specified, as well as keyword arguments for the geometry of the probe
    cross sections such as slit settings slits and T to define the angular divergence and dLoL to define the
    wavelength resolution.
probe (**kw)
    Simulate a measurement probe.
    Returns a probe with Q, angle, wavelength and the associated uncertainties, but not any data.
    You can override instrument parameters using key=value. In particular, slit settings slits and T define the
    angular divergence and dLoL defines the wavelength resolution.
radiation = 'neutron'
resolution (L, dL, **kw)
    Return the resolution of the measurement. Needs T, L, dL specified as keywords.
sample_broadening = 0
sample_width = 10000000000.0
simulate (sample, uncertainty=1, **kw)
    Simulate a run with a particular sample.

Parameters

    sample    [Stack] Reflectometry model
    T         [[float | °] List of angles to be measured, such as [0.15,0.4,1,2].
    slits     [[float] or ([float,float]) | mm] Slit settings for each angle.
    uncertainty = 1  [float or [float] | %] Incident intensity is set so that the median dR/R is equal
to uncertainty, where R is the idealized reflectivity of the sample.
    dLoL = 0.02: float  Wavelength resolution
    normalize = True  [boolean] Whether to normalize the intensities
    theta_offset = 0  [float | °] Sample alignment error
    background = 0  [float] Background counts per incident neutron (background is assumed to
                  be independent of measurement geometry).
    back_reflectivity = False  [boolean] Whether beam travels through incident medium or
                                through substrate.
    back_absorption = 1  [float] Absorption factor for beam traveling through substrate. Only
                        needed for back reflectivity measurements.
slits = None
slits_above = None
slits_at_Tlo = None
slits_below = None
\[
\text{wavelength} = (1.8, 14)
\]

```python
ref1d.snsdata.QRL_to_data(instrument, header, data)
```

Convert data to T.L.R

```python
class refl1d.snsdata.SNSData
    Bases: object
    load(filename, **kw)
```

```python
ref1d.snsdata.TOFC_to_data(instrument, header, data)
```

Convert TOF data to neutron probe.

Wavelength is set from the average of the times at the edges of the bins, not the average of the wavelengths. Wavelength resolution is set assuming the wavelength at the edges of the bins defines the full width at half maximum.

The correct answer is to look at the wavelength distribution within the bin including effects of pulse width and intensity as a function wavelength and use that distribution, or a gaussian approximation thereof, when computing the resolution effects.

```python
ref1d.snsdata.boltzmann_feather(L, counts=100000, range=None)
```

Return expected intensity as a function of wavelength given the TOF feather range and the total number of counts.

TOF feather is approximately a Boltzmann distribution with Gaussian convolution. The following looks pretty enough; don’t know how well it corresponds to the actual SNS feather.

```python
ref1d.snsdata.has_columns(header, v)
```

```python
ref1d.snsdata.intensity_from_spline(Lrange, dLoL, feather)
```

```python
ref1d.snsdata.load(filename, instrument=None, **kw)
```

Return a probe for SNS data.

```python
ref1d.snsdata.parse_sns_file(filename)
```

Parse SNS reduced data, returning `header` and `data`.

- `header` dictionary of fields such as ‘data’, ‘title’, ‘instrument’
- `data` 2D array of data

```python
ref1d.snsdata.write_file(filename, probe, original=None, date=None, title=None, notes=None, run=None, charge=None)
```

Save probe as SNS reduced file.

### 4.25 staj - Staj File

<table>
<thead>
<tr>
<th><strong>MlayerMagnetic</strong></th>
<th>Model definition used by GJ2 program.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MlayerModel</strong></td>
<td>Model definition used by MLayer program.</td>
</tr>
</tbody>
</table>

Read and write staj files

Staj files are the model files for the mlayer and g\text{j}2 programs, which are used as the calculation engine for the reflpak suite. Mlayer supports unpolarized beam with multilayer models, and has files ending in \textit{.staj}. GJ2 supports polarized beam without multilayer models, and has files ending in \textit{.sta}.

```python
class refl1d.staj.MlayerMagnetic(**kw)
    Bases: object
    Model definition used by GJ2 program.
```

4.25. staj - Staj File 137
Attributes:

Q values and reflectivity come from a data file with Q, R, dR or from simulation with linear spacing from Qmin to Qmax in equal steps:

- **data_file** base name of the data file, or None if this is simulation only
- **active_xsec** active cross sections (usually ‘abcd’ for all cross sections)
- **Qmin, Qmax, num_Q** for simulation, Q sample points

Resolution is defined by wavelength and by incident angle:

- **wavelength**, **wavelength_dispersion**, **angular_divergence** resolution is calculated as $\frac{\Delta Q}{Q} = \frac{\Delta \lambda}{\lambda} + \frac{\Delta \theta}{\theta}$

Additional beam parameters correct for intensity, background and possibly guide field angle:

- **intensity**, **background** incident beam intensity and sample background
- **guide_angle** angle of the guide field

Unlike pure structural models, magnetic models are in one large section with no repeats. The single parameter is the number of layers, which is implicit in the length of the layer data and does not need to be an explicit attribute.

Interfaces are split into discrete steps according to a profile, either error function or hyperbolic tangent. For sharp interfaces which do not overlap within a layer, the interface is broken into a fixed number of slabs with slabs having different widths, but equal changes in height. For broad interfaces, the whole layer is split into the same fixed number of slabs, but with each slab having the same width. The following attributes are used:

- **roughness_steps** number of roughness steps (13 is coarse; 51 is fine)
- **roughness_profile** roughness profile is either ‘E’ for error function or ‘H’ for tanh

Layers have thickness, interface roughness and real and imaginary scattering length density (SLD). Roughness is stored in the file using full width at half maximum (FWHM) for the given profile type. For convenience, roughness can also be set or queried using a 1-$\sigma$ equivalent roughness on an error function profile. Regardless, layer parameters are represented as vectors with one entry for each top, middle and bottom layer using the following attributes:

- **thickness**, **roughness** [float | Å] layer thickness and FWHM roughness
- **rho, irho** [float, float | 16$\pi$ρ, 2$\lambda\rho_i$] complex scattering length density
- **mthickness, mroughness** [float | Å] magnetic thickness and roughness
- **mrho** [float | 16$\pi\rho_M$] magnetic scattering length density
- **mtheta** [float | °] magnetic angle
- **sigma_roughness, sigma_mroughness** [float | Å] computed 1-$\sigma$ equivalent roughness for erf profile

The conversion from stored $16\pi\rho, 2\lambda\rho_i$ to in memory $10^6\rho, 10^6\rho_i$ happens automatically on read/write.

The layers are ordered from surface to substrate.

Additional attributes are as follows:

- **fitpars** individual fit parameter numbers
- **constraints** constraints between layers
- **output_file** name of the output file
These can be safely ignored, except perhaps if you want to try to compile the constraints into something that can be used by your system.

**Methods:**

model = MlayerMagnetic(attribute=value, ...)

Construct a new MLayer model with the given attributes set.

model = MlayerMagnetic.load(filename)

Construct a new MLayer model from a sta file.

model.set(attribute=value, ...)

Replace a set of attribute values.

model.fit_resolution(Q,dQ)

Choose the best resolution parameters to match the given Q,dQ resolution. Returns the object so that calls can be chained.

model.resolution(Q)

Return the resolution at Q for the current resolution parameters.

model.save(filename)

Write the model to the given named file. Raises ValueError if the model is invalid.

**Constructing new files:**

Staj files can be constructed directly. The MlayerModel constructor can accept all data attributes as key word arguments. Models require at least `data_file`, `wavelength`, `thickness`, `roughness` and `rho`. Resolution parameters can be set using `model.fit_resolution(Q,dQ)`. Everything else has reasonable defaults.

**FWHMresolution(Q)**

Return the resolution at Q for mlayer with the current settings for wavelength, wavelength divergence and angular divergence.

Resolution is full-width at half maximum (FWHM), not $1-\sigma$.

$Q_{\text{max}} = 0.5$

$Q_{\text{min}} = 0$

`active_xsec = 'abcd'`

`angular_divergence = 0.001`

`background = 0`

`constraints = ''`

`data_file = ''`

`fit_FWHMresolution(Q, dQ, weight=1)`

Choose the best dL and dT to match the resolution dQ.

Given that mlayer uses the following resolution function:

$$\Delta Q_k = (|Q_k|\Delta \lambda + 4\pi \Delta \theta) / \lambda_k$$

we can use a linear system solver to find the optimal $\Delta \lambda$ and $\Delta \theta$ across our dataset from the overdetermined system:

$$[|Q_k| / \lambda_k, 4\pi / \lambda_k][\Delta \lambda, \Delta \theta]^T = \Delta Q_k$$
If weights are provided (e.g., $\Delta R_k/R_k$), then weigh each point during the fit.

Given that the experiment is often run with fixed slits at the start and end, you may choose to match the resolution across the entire $Q$ range, or instead restrict it to just the region where the slits are opening. You will generally want to get the resolution correct at the critical edge since that’s where it will have the largest effect on the fit.

Returns the object so that operations can be chained.

```python
fitpars = []
guide_angle = 270
intensity = 1
irho = None
classmethod load(filename)
    Load a staj file, returning an MlayerModel object
mrho = None
mroughness = None
mtheta = None
mthickness = None
num_Q = 200
output_file = ''
rho = None
roughness = None
roughness_profile = 'E'
roughness_steps = 13
save(filename)
    Save the staj file
set(**kw)
sigma_mroughness
sigma_roughness
thickness = None
wavelength = 1
wavelength_dispersion = 0.01
```

**class refl1d.staj.MlayerModel(**kw)

**Bases:** object

Model definition used by MLayer program.

**Attributes:**

- Q values and reflectivity come from a data file with Q, R, dR or from simulation with linear spacing from Qmin to Qmax in equal steps:
  
  - data_file: name of the data file, or None if this is simulation only
  
  - Qmin, Qmax, num_Q: for simulation, Q sample points
Resolution is defined by wavelength and by incident angle:

\[ \Delta Q/Q = \Delta \lambda/\lambda + \Delta \theta/\theta \]

Additional beam parameters correct for intensity, background and possibly sample alignment:

- intensity, background: incident beam intensity and sample background
- theta_offset: alignment angle correction

The model is defined in terms of layers, with three sections. The top and bottom section correspond to the fixed layers at the surface and the substrate. The middle section layers can be repeated an arbitrary number of times, as defined by the number of repeats attribute. The attributes defining the sections are:

- num_top, num_middle, num_bottom: section sizes
- num_repeats: number of times middle section repeats

Interfaces are split into discrete steps according to a profile, either error function or hyperbolic tangent. For sharp interfaces which do not overlap within a layer, the interface is broken into a fixed number of slabs with slabs having different widths, but equal changes in height. For broad interfaces, the whole layer is split into the same fixed number of slabs, but with each slab having the same width. The following attributes are used:

- roughness_steps: number of roughness steps (13 is coarse; 51 is fine)
- roughness_profile: roughness profile is either ‘E’ for error function or ‘H’ for tanh

Layers have thickness, interface roughness and real and imaginary scattering length density (SLD). Roughness is stored in the file using full width at half maximum (FWHM) for the given profile type. For convenience, roughness can also be set or queried using a 1-\(\sigma\) equivalent roughness on an error function profile. Regardless, layer parameters are represented as vectors with one entry for each top, middle and bottom layer using the following attributes:

- thickness, roughness: [float | Å] layer thickness and FWHM roughness
- rho, irho, incoh: [float | 10^{-6}Å^{-2}] complex coherent \(\rho + j\rho_i\) and incoherent SLD

Computed attributes are provided for convenience:

- sigma_roughness: [float | Å] 1-\(\sigma\) equivalent roughness for erf profile
- mu: absorption cross section (2*wavelength*irho + incoh)

**Note:** The staj files store SLD as \(16\pi\rho, 2\lambda\rho_i\) with an additional column of 0 for magnetic SLD. This conversion happens automatically on read/write. The incoherent cross section is assumed to be zero.

The layers are ordered from surface to substrate.

Additional attributes are as follows:

- fitpars: individual fit parameter numbers
- constraints: constraints between layers
- output_file: name of the output file

These can be safely ignored, except perhaps if you want to try to compile the constraints into something that can be used by your system.

**Methods:**

```python
model = MlayerModel(attribute=value, ...)
```

Construct a new MLayer model with the given attributes set.
model = MlayerModel.load(filename)
      Construct a new MLayer model from a staj file.
model.set(attribute=value, ...)
      Replace a set of attribute values.
model.fit_resolution(Q,dQ)
      Choose the best resolution parameters to match the given Q,dQ resolution. Returns the object so that
      calls can be chained.
model.resolution(Q)
      Return the resolution at Q for the current resolution parameters.
model.split_sections()
      Assign top, middle, bottom and repeats to distribute the layers across sections. Returns the object so
      that calls can be chained.
model.save(filename)
      Write the model to the given named file. Raises ValueError if the model is invalid.

Constructing new files:

Staj files can be constructed directly. The MlayerModel constructor can accept all data attributes as key word
arguments. Models require at least data_file, wavelength, thickness, roughness and rho. Resolution parameters
can be set using model.fit_resolution(Q,dQ). Section sizes can be set using model.split_sections(). Everything
else has reasonable defaults.

FWHMresolution(Q)
      Return the resolution at Q for mlayer with the current settings for wavelength, wavelength divergence and
      angular divergence.

Resolution is full-width at half maximum (FWHM), not 1-σ.

Qmax = 0.5
Qmin = 0
angular_divergence = 0.001
background = 0
constraints = ‘’
data_file = ‘’

fit_FWHMresolution(Q, dQ, weight=1)
      Choose the best dL and dT to match the resolution dQ.
      Given that mlayer uses the following resolution function:

      \[ \Delta Q_k = \frac{(|Q_k| \Delta \lambda + 4\pi \Delta \theta)}{\lambda_k} \]

      we can use a linear system solver to find the optimal \( \Delta \lambda \) and \( \Delta \theta \) across our dataset from the over-
      determined system:

      \[ [|Q_k|/\lambda_k, 4\pi/\lambda_k][\Delta \lambda, \Delta \theta]^T = \Delta Q_k \]

      If weights are provided (e.g., \( \Delta R_k/R_k \)), then weigh each point during the fit.

      Given that the experiment is often run with fixed slits at the start and end, you may choose to match the
      resolution across the entire \( Q \) range, or instead restrict it to just the region where the slits are opening.
You will generally want to get the resolution correct at the critical edge since that’s where it will have the largest effect on the fit.

Returns the object so that operations can be chained.

```python
def fitpars:
    fitpars = []
    incoh = 0
    intensity = 1
    irho = 0

classmethod load(filename):
    Load a staj file, returning an MlayerModel object
    
    mu
    num_Q = 200
    num_bottom = 0
    num_middle = 0
    num_repeats = 1
    num_top = 0
    output_file = ''
    rho = None
    roughness = None
    roughness_profile = 'E'
    roughness_steps = 13

def save(filename):
    Save the staj file

def set(**kw):

    sigma_roughness

    split_sections()
    Split the given set of layers into sections, putting as many layers as possible into the middle section, then the bottom and finally the top.

    Returns the object so that operations can be chained.

    theta_offset = 0
    thickness = None
    wavelength = 1
    wavelength Dispersion = 0.01
```

## 4.26 stajconvert - Staj File Converter

- `fit_all` Set all non-zero parameters to fitted parameters inside the model.
- `load_mlayer` Load a staj file as a model.
### Table 4.26 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mlayer_magnetic_to_model</td>
<td>Convert a loaded sta file to a refl1d experiment.</td>
</tr>
<tr>
<td>mlayer_to_model</td>
<td>Convert a loaded staj file to a refl1d experiment.</td>
</tr>
<tr>
<td>model_to_mlayer</td>
<td>Return an mlayer model based on the slab stack.</td>
</tr>
<tr>
<td>save_mlayer</td>
<td>Save a model to a staj file.</td>
</tr>
</tbody>
</table>

Convert staj files to Refl1D models

```python
refl1d.stajconvert.fit_all(M, pmp=20)
Set all non-zero parameters to fitted parameters inside the model.
```

```python
refl1d.stajconvert.load_mlayer(filename, fit_pmp=0, name=None, layers=None)
Load a staj file as a model.
```

```python
refl1d.stajconvert.mlayer_magnetic_to_model(sta, name=None, layers=None)
Convert a loaded sta file to a refl1d experiment.
Returns a new experiment
```

```python
refl1d.stajconvert.mlayer_to_model(staj, name=None, layers=None)
Convert a loaded staj file to a refl1d experiment.
Returns a new experiment
```

```python
refl1d.stajconvert.model_to_mlayer(model, datafile)
Return an mlayer model based on the a slab stack.
Raises TypeError if model cannot be stored as a staj file.
```

```python
refl1d.stajconvert.save_mlayer(experiment, filename, datafile=None)
Save a model to a staj file.
```

### 4.27 stitch - Overlapping reflectivity curve stitching

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>poisson_average</td>
<td>Compute the poisson average of y/dy using a set of data points.</td>
</tr>
<tr>
<td>stitch</td>
<td>Stitch together multiple measurements into one.</td>
</tr>
</tbody>
</table>

Data stitching.

Join together datasets yielding unique sorted x.

```python
refl1d.stitch.poisson_average(xxydyw)
Compute the poisson average of y/dy using a set of data points.
The returned x,dy is the weighted average of the inputs:

\[
x = \frac{\text{sum}(x*I)}{\text{sum}(I)}
\]

\[
dx = \frac{\text{sum}(dx*I)}{\text{sum}(I)}
\]

The returned y,dy use Poisson averaging:

\[
w = \text{sum}(y/dy^2)
\]

\[
y = \frac{\text{sum}((y/dy)^2)}{w}
\]

\[
dy = \sqrt{\frac{y}{w}}
\]

The above formula gives the expected result for combining two measurements, assuming there is no uncertainty in the monitor.
measure N counts during M monitors
rate: \( r = \frac{N}{M} \)
rate uncertainty: \( dr = \sqrt{\frac{N}{M}} \)
weighted rate: \( \frac{r}{dr^2} = \frac{(N/M)}{(N/M^2)} = M \)
weighted rate squared: \( \frac{r^2}{dr^2} = \frac{(N^2/M^2)}{(N/M^2)} = N \)

for two measurements \( Na, Nb \)
\[ w = \frac{ra/dra^2 + rb/dbc^2}{Ma + Mb} \]
\[ y = \left( \frac{(ra/dra)^2 + (rb/dbc)^2}{w} \right) = \frac{(Na + Nb)}{(Ma + Mb)} \]
\[ dy = \sqrt{\frac{y}{w}} = \sqrt{\frac{(Na + Nb)}{(Ma + Mb)}} \]

This formula isn’t strictly correct when applied to values which have been scaled, for example to account for an attenuator in the counting system.

reffi1d.stitch.stitch(data, same_x=0.001, same_dx=0.001)
Stitch together multiple measurements into one.

\( data \) a list of datasets with x,dx,y,dy attributes \( same_x \) minimum point separation (default is 0.001). \( same_dx \) minimum change in resolution that may be averaged (default is 0.001).

WARNING: the returned x values may be data dependent, with two measured sets having different x after stitching, even though the measurement conditions are identical!!

Either add an intensity weight to the datasets:

probe.I = slitscan

or use interpolation if you need to align two stitched scans:
\[
\begin{align*}
  x1,dx1,y1,dy1 &= \text{stitch}([a1,b1,c1,d1]) \\
  x2,dx2,y2,dy2 &= \text{stitch}([a2,b2,c2,d2]) \\
  x2[0],x2[-1] &= x1[0],x1[-1] \quad \# \text{Force matching end points} \\
  y2 &= \text{numpy.interp}(x1,x2,y2) \\
  dy2 &= \text{numpy.interp}(x1,x2,dy2) \\
  x2 &= x1
\end{align*}
\]

WARNING: the returned dx value underestimates the true x, depending on the relative weights of the averaged data points.

### 4.28 support - Environment support

get_data_path
Locate the examples directory.
sample_data
Support files for the application.
This includes tools to help with testing, documentation, command line parsing, etc. which are specific to this application, rather than general utilities.

reffi1d.support.get_data_path()
Locate the examples directory.
reffi1d.support.sample_data(file)
4.29 util - Miscellaneous functions

```python
refl1d.util.merge_ends(w, p, tol=0.001)
```

join the leading and trailing ends of the profile together so fewer slabs are required and so that gaussian roughness can be used.

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abeles</td>
<td>Optical matrix form of the reflectivity calculation.</td>
</tr>
<tr>
<td>cheby</td>
<td>Freeform modeling with Chebyshev polynomials</td>
</tr>
<tr>
<td>dist</td>
<td>Inhomogeneous samples</td>
</tr>
<tr>
<td>errors</td>
<td>Visual representation of model uncertainty.</td>
</tr>
<tr>
<td>experiment</td>
<td>Experiment definition</td>
</tr>
<tr>
<td>fitplugin</td>
<td>Reflectivity plugin for fitting GUI.</td>
</tr>
<tr>
<td>freeform</td>
<td>Freeform modeling with B-Splines</td>
</tr>
<tr>
<td>fresnel</td>
<td>Pure python Fresnel reflectivity calculator.</td>
</tr>
<tr>
<td>garefl</td>
<td>Load garefl models into refl1d.</td>
</tr>
<tr>
<td>instrument</td>
<td>Reflectometry instrument definitions.</td>
</tr>
<tr>
<td>magnetism</td>
<td>Magnetic modeling for 1-D reflectometry.</td>
</tr>
<tr>
<td>material</td>
<td>Reflectometry materials.</td>
</tr>
<tr>
<td>materialdb</td>
<td>Common materials in reflectometry experiments along with densities.</td>
</tr>
<tr>
<td>model</td>
<td>Reflectometry models</td>
</tr>
<tr>
<td>mono</td>
<td>Monotonic spline modeling for free interfaces</td>
</tr>
<tr>
<td>names</td>
<td>Exported names</td>
</tr>
<tr>
<td>ncnrdata</td>
<td>NCNR data loaders</td>
</tr>
<tr>
<td>polymer</td>
<td>Layer models for polymer systems.</td>
</tr>
<tr>
<td>probe</td>
<td>Experimental probe.</td>
</tr>
<tr>
<td>profile</td>
<td>Scattering length density profile.</td>
</tr>
<tr>
<td>reflectivity</td>
<td>Basic reflectometry calculations</td>
</tr>
<tr>
<td>reflmodule</td>
<td>Reflectometry C Library</td>
</tr>
<tr>
<td>resolution</td>
<td>Resolution calculations</td>
</tr>
<tr>
<td>snsdata</td>
<td>SNS data loaders</td>
</tr>
<tr>
<td>staj</td>
<td>Read and write staj files</td>
</tr>
<tr>
<td>stajconvert</td>
<td>Convert staj files to Refl1D models</td>
</tr>
<tr>
<td>stitch</td>
<td>Data stitching.</td>
</tr>
<tr>
<td>support</td>
<td>Support files for the application.</td>
</tr>
<tr>
<td>util</td>
<td>Support files for the application.</td>
</tr>
</tbody>
</table>
refl1d.abeles, 53
refl1d.cheby, 54
refl1d.dist, 56
refl1d.errors, 58
refl1d.experiment, 60
refl1d.fitplugin, 64
refl1d.freeform, 65
refl1d.fresnel, 68
refl1d.garefl, 68
refl1d.instrument, 69
refl1d.magnetism, 75
refl1d.material, 77
refl1d.materialdb, 80
refl1d.model, 81
refl1d.mono, 84
refl1d.names, 85
refl1d.ncnrdata, 85
refl1d.polymer, 98
refl1d.probe, 103
refl1d.profile, 124
refl1d.reflectivity, 127
refl1d.reflectmodule, 129
refl1d.resolution, 130
refl1d.snsdata, 133
refl1d.staj, 137
refl1d.stajconvert, 144
refl1d.stitch, 144
refl1d.support, 145
refl1d.util, 146
active_xsec (refl1d.staj.MlayerMagnetic attribute), 139
add() (refl1d.model.Stack method), 82
add_magnetism() (refl1d.profile.Microslabs method), 125
Aguide (refl1d.probe.NeutronProbe attribute), 103
Aguide (refl1d.probe.Probe attribute), 110
Aguide (refl1d.probe.ProbeSet attribute), 113
Aguide (refl1d.probe.QProbe attribute), 117
Aguide (refl1d.probe.XrayProbe attribute), 120
align_profiles() (in module refl1d.errors), 59
alignment_uncertainty() (refl1d.probe.NeutronProbe method), 103
alignment_uncertainty() (refl1d.probe.Probe static method), 110
alignment_uncertainty() (refl1d.probe.ProbeSet method), 113
alignment_uncertainty() (refl1d.probe.QProbe method), 117
alignment_uncertainty() (refl1d.probe.XrayProbe method), 120
amplitude() (refl1d.experiment.Experiment method), 60
amplitude() (refl1d.experiment.MixedExperiment method), 63
ANDR (class in refl1d.ncnrdata), 86
angular_divergence (refl1d.staj.MlayerMagnetic attribute), 139
angular_divergence (refl1d.staj.MlayerModel attribute), 142
append() (refl1d.profile.Microslabs method), 125
apply_beam() (refl1d.probe.NeutronProbe method), 103
apply_beam() (refl1d.probe.PolarizedNeutronProbe method), 106
apply_beam() (refl1d.probe.PolarizedQProbe method), 108
apply_beam() (refl1d.probe.Probe method), 111
apply_beam() (refl1d.probe.ProbeSet method), 114
apply_beam() (refl1d.probe.QProbe method), 117
apply_beam() (refl1d.probe.XrayProbe method), 120
background (refl1d.staj.MlayerMagnetic attribute), 139
background (refl1d.staj.MlayerModel attribute), 142
BaseMagnetism (class in refl1d.magnetism), 75
binedges() (in module refl1d.resolution), 131
bins() (in module refl1d.resolution), 131
binwidths() (in module refl1d.resolution), 131
blend() (in module refl1d.profile), 127
boltzmann_feather() (in module refl1d.snsdata), 137
build_mag_profile() (in module refl1d.profile), 127
build_profile() (in module refl1d.profile), 127
bymass() (refl1d.material.Mixture class method), 79
byvolume() (refl1d.material.Mixture class method), 79
calc() (in module refl1d.abeles), 53
calc_dT() (refl1d.instrument.Monochromatic method), 71
calc_dT() (refl1d.instrument.Pulsed method), 73
calc_dT() (refl1d.ncnrdata.ANDR method), 86
calc_dT() (refl1d.ncnrdata.MAGIK method), 88
calc_dT() (refl1d.ncnrdata.NG1 method), 90
calc_dT() (refl1d.ncnrdata.NG7 method), 92
calc_dT() (refl1d.ncnrdata.PBR method), 93
calc_dT() (refl1d.ncnrdata.XRay method), 95
calc_dT() (refl1d.snsdata.Liquids method), 134
calc_dT() (refl1d.snsdata.Magnetic method), 135
calc_errors() (in module refl1d.errors), 59
calc_errors() (in module refl1d.fitplugin), 65
calc_Q (refl1d.probe.NeutronProbe attribute), 104
calc_Q (refl1d.probe.PolarizedNeutronProbe attribute), 106
calc_Q (refl1d.probe.PolarizedQProbe attribute), 108
calc_Q (refl1d.probe.Probe attribute), 111
calc_Q (refl1d.probe.ProbeSet attribute), 114
calc_Q (refl1d.probe.QProbe attribute), 117
calc_Q (refl1d.probe.XRayProbe attribute), 120
calc_slits() (refl1d.instrument.Monochromatic method), 72
calc_slits() (refl1d.instrument.Pulsed method), 73
calc_slits() (refl1d.ncnrdata.ANDR method), 87
calc_slits() (refl1d.ncnrdata.MAGIK method), 88
calc_slits() (refl1d.ncnrdata.NG1 method), 90
calc_slits() (refl1d.ncnrdata.NG7 method), 92
calc_slits() (refl1d.ncnrdata.PBR method), 94
calc_slits() (refl1d.ncnrdata.XRay method), 96
calc_slits() (refl1d.snsdata.Liquids method), 134
calc_slits() (refl1d.snsdata.Magnetic method), 135
ChebyVF (class in refl1d.cheby), 54
clear() (refl1d.material.ProbeCache method), 80
clear() (refl1d.profile.Microslabs method), 125
compute_limited_sigma() (in module refl1d.profile), 127
constraints (refl1d.staj.MlayerMagnetic attribute), 139
constraints (refl1d.staj.MlayerModel attribute), 142
constraints() (refl1d.cheby.ChebyVF method), 54
constraints() (refl1d.cheby.FreeformCheby method), 55
constraints() (refl1d.freeform.FreeformInterface01 method), 67
constraints() (refl1d.freeform.FreeInterface method), 66
constraints() (refl1d.model.Layer method), 81
constraints() (refl1d.model.Repeat method), 81
constraints() (refl1d.model.Slab method), 82
constraints() (refl1d.model.Stack method), 82
constraints() (refl1d.mono.FreeInterface method), 84
constraints() (refl1d.mono.FreeLayer method), 84
constraints() (refl1d.polymer.EndTetheredPolymer method), 101
constraints() (refl1d.polymer.PolymerBrush method), 99
constraints() (refl1d.polymer.PolymerMushroom method), 100
constraints() (refl1d.polymer.VolumeProfile method), 102
convolve() (in module refl1d.reflectivity), 129
convolve() (in module refl1d.reflectmodule), 129
convolve_sampled() (in module refl1d.reflectmodule), 129
critical_edge() (refl1d.probe.NeutronProbe method), 104
critical_edge() (refl1d.probe.ProbeSet method), 114
critical_edge() (refl1d.probe.QProbe method), 117
critical_edge() (refl1d.probe.XrayProbe method), 120
D
d_detector (refl1d.ncnrdata.NG7 attribute), 92
d_detector (refl1d.ncnrdata.XRay attribute), 96
d_moderator (refl1d.snsdata.Liquids attribute), 134
ds_1 (refl1d.instrument.Monoenergetic attribute), 72
ds_1 (refl1d.instrument.Pulsed attribute), 74
ds_1 (refl1d.ncnrdata.ANDR attribute), 87
ds_1 (refl1d.ncnrdata.MAGIK attribute), 89
ds_1 (refl1d.ncnrdata.NG1 attribute), 91
ds_1 (refl1d.ncnrdata.NG7 attribute), 92
ds_1 (refl1d.ncnrdata.PBR attribute), 94
ds_1 (refl1d.ncnrdata.XRay attribute), 96
ds_1 (refl1d.snsdata.Liquids attribute), 134
ds_1 (refl1d.snsdata.Magnetic attribute), 135
ds_2 (refl1d.instrument.Monoenergetic attribute), 72
ds_2 (refl1d.instrument.Pulsed attribute), 74
ds_2 (refl1d.ncnrdata.ANDR attribute), 87
ds_2 (refl1d.ncnrdata.MAGIK attribute), 89
ds_2 (refl1d.ncnrdata.NG1 attribute), 91
ds_2 (refl1d.ncnrdata.NG7 attribute), 92
ds_2 (refl1d.ncnrdata.PBR attribute), 94
ds_2 (refl1d.ncnrdata.XRay attribute), 96
ds_2 (refl1d.snsdata.Liquids attribute), 134
ds_2 (refl1d.snsdata.Magnetic attribute), 135
d_s1 (refl1d.snsdata.Liquids attribute), 134

Index
finalize() (refl1d.profile.Microslabs method), 125
find() (refl1d.cheby.ChebyVF method), 54
find() (refl1d.cheby.FreeformCheby method), 55
find() (refl1d.freeform.FreeformInterface01 method), 67
find() (refl1d.freeform.FreeInterface method), 66
find() (refl1d.model.Layer method), 83
find() (refl1d.model.Repeat method), 81
find() (refl1d.model.Slab method), 82
find() (refl1d.model.Stack method), 82
find() (refl1d.mono.FreeInterface method), 84
find() (refl1d.mono.FreeLayer method), 84
find() (refl1d.polymer.EndTetheredPolymer method), 101
find() (refl1d.polymer.PolymerBrush method), 99
find() (refl1d.polymer.PolymerMushroom method), 100
find() (refl1d.polymer.VolumeProfile method), 102
find_xsec() (in module refl1d.ncnrdata), 97
fit_all() (in module refl1d.stajconvert), 144
fit_FWHMresolution() (refl1d.staj.MlayerMagnetic method), 139
fit_FWHMresolution() (refl1d.staj.MlayerModel method), 142
fitby() (refl1d.material.Material method), 78
fitpars (refl1d.staj.MlayerMagnetic attribute), 140
fitpars (refl1d.staj.MlayerModel attribute), 143
fixed_slits (refl1d.instrument.Monochromatic attribute), 72
fixed_slits (refl1d.instrument.Pulsed attribute), 74
fixed_slits (refl1d.ncnrdata.ANDR attribute), 87
fixed_slits (refl1d.ncnrdata.MAGIK attribute), 89
fixed_slits (refl1d.ncnrdata.NG1 attribute), 91
fixed_slits (refl1d.ncnrdata.NG7 attribute), 92
fixed_slits (refl1d.ncnrdata.PBR attribute), 94
fixed_slits (refl1d.ncnrdata.XRay attribute), 96
fixed_slits (refl1d.snsdata.Liquids attribute), 134
fixed_slits (refl1d.snsdata.Magnetic attribute), 135
format_parameters() (refl1d.dist.DistributionExperiment method), 56
format_parameters() (refl1d.experiment.Experiment method), 60
format_parameters() (refl1d.experiment.ExperimentBase method), 62
format_parameters() (refl1d.experiment.MixedExperiment method), 63
FreeformCheby (class in refl1d.cheby), 55
FreeformInterface01 (class in refl1d.freeform), 67
FreeInterface (class in refl1d.freeform), 65
FreeInterface (class in refl1d.mono), 84
FreeLayer (class in refl1d.freeform), 66
FreeLayer (class in refl1d.mono), 84
FreeMagnetism (class in refl1d.magnetism), 76
Fresnel (class in refl1d.fresnel), 68
fresnel() (refl1d.probe.NeutronProbe method), 104
fresnel() (refl1d.probe.PolarizedNeutronProbe method), 106
fresnel() (refl1d.probe.PolarizedQProbe method), 108
fresnel() (refl1d.probe.Probe method), 111
fresnel() (refl1d.probe.ProbeSet method), 114
fresnel() (refl1d.probe.QProbe method), 118
fresnel() (refl1d.probe.XrayProbe method), 120
FWHM2sigma() (in module refl1d.resolution), 130
FWHMresolution() (refl1d.staj.MlayerMagnetic method), 139
FWHMresolution() (refl1d.staj.MlayerModel method), 142
get_data_path() (in module refl1d.support), 145
guide_angle (refl1d.staj.MlayerMagnetic attribute), 140
has_columns() (in module refl1d.snsdata), 137
incoh (refl1d.staj.MlayerModel attribute), 143
inflections() (in module refl1d.mono), 85
insert() (refl1d.model.Stack method), 82
instrument (refl1d.instrument.Monochromatic attribute), 72
instrument (refl1d.instrument.Pulsed attribute), 74
instrument (refl1d.ncnrdata.ANDR attribute), 87
instrument (refl1d.ncnrdata.MAGIK attribute), 89
instrument (refl1d.ncnrdata.NG1 attribute), 91
instrument (refl1d.ncnrdata.NG7 attribute), 92
instrument (refl1d.ncnrdata.PBR attribute), 94
instrument (refl1d.ncnrdata.XRay attribute), 96
instrument (refl1d.snsdata.Liquids attribute), 134
instrument (refl1d.snsdata.Magnetic attribute), 136
intensity (refl1d.staj.MlayerMagnetic attribute), 140
intensity (refl1d.staj.MlayerModel attribute), 143
intensity_from_spline() (in module refl1d.snsdata), 137
interface (refl1d.cheby.ChebyVF attribute), 54
interface (refl1d.cheby.FreeformCheby attribute), 55
interface (refl1d.freeform.FreeformInterface01 attribute), 67
interface (refl1d.freeform.FreeInterface attribute), 66
interface (refl1d.freeform.FreeLayer attribute), 66
interface (refl1d.model.Layer attribute), 83
interface (refl1d.model.Repeat attribute), 81
interface (refl1d.model.Slab attribute), 82
interface (refl1d.model.Stack attribute), 82
interface (refl1d.mono.FreeInterface attribute), 84
interface (refl1d.mono.FreeLayer attribute), 85
interface (refl1d.polymer.EndTetheredPolymer attribute), 101
interface (refl1d.polymer.PolymerBrush attribute), 99
interface (refl1d.polymer.PolymerMushroom attribute),
  100
interface (refl1d.polymer.VolumeProfile attribute), 102
interface() (refl1d.profile.Microslabs method), 126
irho (refl1d.polymer.VolumeProfile attribute), 102
irho (refl1d.staj.MlayerMagnetic attribute), 140
irho (refl1d.staj.MlayerModel attribute), 143
is_reset() (refl1d.dist.DistributionExperiment method), 56
is_reset() (refl1d.experiment.Experiment method), 60
is_reset() (refl1d.experiment.ExperimentBase method), 62
is_reset() (refl1d.experiment.MixedExperiment method), 63
ismagnetic (refl1d.cheby.ChebyVF attribute), 54
ismagnetic (refl1d.cheby.FreeformCheby attribute), 55
ismagnetic (refl1d.experiment.Experiment attribute), 60
ismagnetic (refl1d.freeform.FreeformInterface01 attribute), 67
ismagnetic (refl1d.freeform.FreeInterface attribute), 66
ismagnetic (refl1d.model.Layer attribute), 83
ismagnetic (refl1d.model.Repeat attribute), 81
ismagnetic (refl1d.model.Slab attribute), 82
ismagnetic (refl1d.model.Stack attribute), 83
ismagnetic (refl1d.mono.FreeInterface attribute), 84
ismagnetic (refl1d.mono.FreeLayer attribute), 85
ismagnetic (refl1d.polymer.EndTetheredPolymer attribute), 101
ismagnetic (refl1d.polymer.PolymerBrush attribute), 99
ismagnetic (refl1d.polymer.PolymerMushroom attribute), 100
ismagnetic (refl1d.polymer.VolumeProfile attribute), 102
ismagnetic (refl1d.profile.Microslabs attribute), 126
layer() (refl1d.probe.NeutronProbe method), 104
layer() (refl1d.probe.Probe method), 111
layer() (refl1d.probe.ProbeSet method), 114
layer() (refl1d.probe.QProbe method), 118
layer() (refl1d.probe.XrayProbe method), 121
Layer (class in refl1d.model), 83
layer_parameters() (refl1d.cheby.ChebyVF method), 54
layer_parameters() (refl1d.cheby.FreeformCheby method), 55
layer_parameters() (refl1d.freeform.FreeformInterface01 method), 67
layer_parameters() (refl1d.freeform.FreeInterface method), 66
layer_parameters() (refl1d.freeform.FreeLayer method), 66
layer_parameters() (refl1d.model.Layer method), 83
layer_parameters() (refl1d.model.Repeat method), 81
layer_parameters() (refl1d.model.Slab method), 82
layer_parameters() (refl1d.model.Stack method), 83
layer_parameters() (refl1d.mono.FreeInterface method), 84
layer_parameters() (refl1d.mono.FreeLayer method), 85
layer_parameters() (refl1d.polymer.EndTetheredPolymer method), 101
layer_parameters() (refl1d.polymer.PolymerBrush method), 99
layer_parameters() (refl1d.polymer.PolymerMushroom method), 100
layer_parameters() (refl1d.polymer.VolumeProfile method), 102
layer_thickness() (in module refl1d.polymer), 103
Liquids (class in refl1d.snsdata), 133
load() (in module refl1d.garefl), 69
load() (in module refl1d.ncnrdata), 97
load() (in module refl1d.snsdata), 137
load() (refl1d.ncnrdata.ANDR method), 87
load() (refl1d.ncnrdata.MAGIK method), 89
load() (refl1d.ncnrdata.NCNRData method), 90
load() (refl1d.ncnrdata.NG1 method), 91
load() (refl1d.ncnrdata.NG7 method), 92
load() (refl1d.ncnrdata.PBR method), 94
load() (refl1d.ncnrdata.XRay method), 96
load() (refl1d.snsdata.Liquids method), 134
load() (refl1d.snsdata.Magnetic method), 136
load() (refl1d.snsdata.SNSData method), 137
load() (refl1d.staj.MlayerMagnetic class method), 140
load() (refl1d.staj.MlayerModel class method), 143
load4() (in module refl1d.probe), 123
load_magnetic() (in module refl1d.ncnrdata), 97
load_magnetic() (refl1d.ncnrdata.ANDR method), 87
load_magnetic() (refl1d.ncnrdata.MAGIK method), 89
load_magnetic() (refl1d.ncnrdata.NCNRData method), 90
load_magnetic() (refl1d.ncnrdata.NG1 method), 91
load_magnetic() (refl1d.ncnrdata.NG7 method), 92
load_magnetic() (refl1d.ncnrdata.PBR method), 94
load_magnetic() (refl1d.ncnrdata.XRay method), 96
load() (refl1d.snsdata.Liquids method), 134
load() (refl1d.snsdata.Magnetic method), 136
load() (refl1d.snsdata.SNSData method), 137
load() (refl1d.staj.MlayerMagnetic class method), 140
load() (refl1d.staj.MlayerModel class method), 143
load4() (in module refl1d.probe), 123
log10_to_linear() (refl1d.probe.NeutronProbe method), 104
log10_to_linear() (refl1d.probe.Probe method), 111
log10_to_linear() (refl1d.probe.ProbeSet method), 114
log10_to_linear() (refl1d.probe.QProbe method), 118
log10_to_linear() (refl1d.probe.XrayProbe method), 121
MAGIK (class in refl1d.ncnrdata), 88
Magnetic (class in refl1d.snsdata), 135
magnetic (refl1d.magnetism.FreeMagnetism attribute), 76
magnetic (refl1d.magnetism.MagnetismTwist attribute), 77
magnetic_amplitude() (in module refl1d.reflectivity), 128
plot_log() (refl1d.probe.PolarizedNeutronProbe method), 107
plot_log() (refl1d.probe.PolarizedQProbe method), 109
plot_log() (refl1d.probe.Probe method), 112
plot_log() (refl1d.probe.ProbeSet method), 115
plot_log() (refl1d.probe.QProbe method), 119
plot_log() (refl1d.probe.XrayProbe method), 122
plot_logfresnel() (refl1d.probe.NeutronProbe method), 105
plot_logfresnel() (refl1d.probe.PolarizedNeutronProbe method), 107
plot_logfresnel() (refl1d.probe.PolarizedQProbe method), 109
plot_logfresnel() (refl1d.probe.Probe method), 112
plot_logfresnel() (refl1d.probe.ProbeSet method), 115
plot_logfresnel() (refl1d.probe.QProbe method), 119
plot_logfresnel() (refl1d.probe.XrayProbe method), 122
plot_profile() (refl1d.dist.DistributionExperiment method), 56
plot_profile() (refl1d.experiment.Experiment method), 61
plot_profile() (refl1d.experiment.ExperimentBase method), 62
plot_profile() (refl1d.experiment.MixedExperiment method), 63
plot_Q4() (refl1d.probe.NeutronProbe method), 105
plot_Q4() (refl1d.probe.PolarizedNeutronProbe method), 107
plot_Q4() (refl1d.probe.PolarizedQProbe method), 108
plot_Q4() (refl1d.probe.Probe method), 112
plot_Q4() (refl1d.probe.ProbeSet method), 115
plot_Q4() (refl1d.probe.QProbe method), 118
plot_Q4() (refl1d.probe.XrayProbe method), 121
plot_reflectivity() (refl1d.dist.DistributionExperiment method), 56
plot_reflectivity() (refl1d.experiment.Experiment method), 61
plot_reflectivity() (refl1d.experiment.ExperimentBase method), 62
plot_reflectivity() (refl1d.experiment.MixedExperiment method), 63
plot_residuals() (refl1d.probe.NeutronProbe method), 105
plot_residuals() (refl1d.probe.PolarizedNeutronProbe method), 107
plot_residuals() (refl1d.probe.PolarizedQProbe method), 109
plot_residuals() (refl1d.probe.Probe method), 112
plot_residuals() (refl1d.probe.ProbeSet method), 115
plot_residuals() (refl1d.probe.QProbe method), 119
plot_residuals() (refl1d.probe.XrayProbe method), 122
plot_resolution() (refl1d.probe.NeutronProbe method), 105
plot_resolution() (refl1d.probe.PolarizedNeutronProbe method), 107
plot_resolution() (refl1d.probe.PolarizedQProbe method), 109
plot_resolution() (refl1d.probe.Probe method), 112
plot_resolution() (refl1d.probe.ProbeSet method), 115
plot_resolution() (refl1d.probe.QProbe method), 119
plot_resolution() (refl1d.probe.XrayProbe method), 122
plot_SA() (refl1d.experiment.MixedExperiment method), 63
plot_shift (refl1d.probe.NeutronProbe attribute), 105
plot_shift (refl1d.probe.Probe attribute), 112
plot_shift (refl1d.probe.ProbeSet attribute), 115
plot_shift (refl1d.probe.QProbe attribute), 119
plot_shift (refl1d.probe.XrayProbe attribute), 122
plot_weights() (refl1d.dist.DistributionExperiment method), 56
pm (refl1d.probe.PolarizedNeutronProbe attribute), 107
pm (refl1d.probe.PolarizedQProbe attribute), 109
poisson_average() (in module refl1d.stitch), 144
polarized (refl1d.probe.NeutronProbe attribute), 105
polarized (refl1d.probe.PolarizedNeutronProbe attribute), 107
polarized (refl1d.probe.PolarizedQProbe attribute), 109
polarized (refl1d.probe.Probe attribute), 112
polarized (refl1d.probe.ProbeSet attribute), 115
polarized (refl1d.probe.QProbe attribute), 119
polarized (refl1d.probe.XrayProbe attribute), 122
PolymerBrush (class in refl1d.polymer), 98
PolymerMushroom (class in refl1d.polymer), 99
pp (refl1d.probe.PolarizedNeutronProbe attribute), 107
pp (refl1d.probe.PolarizedQProbe attribute), 109
Probe (class in refl1d.probe), 109
probe() (refl1d.instrument.Monochromatic method), 72
probe() (refl1d.instrument.Pulsed method), 74
probe() (refl1d.ncnrdata.ANDR method), 87
probe() (refl1d.ncnrdata.MAGIK method), 89
probe() (refl1d.ncnrdata.NG1 method), 91
probe() (refl1d.ncnrdata.NG7 method), 93
probe() (refl1d.ncnrdata.PBR method), 94
probe() (refl1d.ncnrdata.XRay method), 96
probe() (refl1d.snsdata.Liquids method), 134
ProbeCache (class in refl1d.material), 80
ProbeSet (class in refl1d.probe), 113
profile() (refl1d.magnetism.FreeMagnetism method), 76
profile() (refl1d.mono.FreeInterface method), 84
profile() (refl1d.mono.FreeLayer method), 85
profile() (refl1d.polymer.EndTetheredPolymer method), 101
profile() (refl1d.polymer.PolymerBrush method), 99
profile() (refl1d.polymer.PolymerMushroom method), 100
profile_shift (refl1d.experiment.Experiment attribute), 61
Pulsed (class in refl1d.instrument), 73

Q
Q (refl1d.probe.NeutronProbe attribute), 103
Q (refl1d.probe.Probe attribute), 110
Q (refl1d.probe.ProbeSet attribute), 113
Q (refl1d.probe.QProbe attribute), 117
Q (refl1d.probe.XrayProbe attribute), 120
QL2T() (in module refl1d.resolution), 130
Qmax (refl1d.staj.MlayerMagnetic attribute), 139
Qmax (refl1d.staj.MlayerModel attribute), 142
Qmeasurement_union() (in module refl1d.probe), 120
Qmin (refl1d.staj.MlayerMagnetic attribute), 139
Qmin (refl1d.staj.MlayerModel attribute), 142
QProbe (class in refl1d.probe), 117
QRL_to_data() (in module refl1d.snsdata), 137
QT2L() (in module refl1d.resolution), 130

R
radiation (refl1d.instrument.Monochromatic attribute), 72
radiation (refl1d.instrument.Pulsed attribute), 74
radiation (refl1d.ncnrdata.ANDR attribute), 88
radiation (refl1d.ncnrdata.MAGIK attribute), 89
radiation (refl1d.ncnrdata.NG1 attribute), 91
radiation (refl1d.ncnrdata.NG7 attribute), 93
radiation (refl1d.ncnrdata.PBR attribute), 95
radiation (refl1d.ncnrdata.XRay attribute), 97
radiation (refl1d.probe.NeutronProbe attribute), 105
radiation (refl1d.probe.XrayProbe attribute), 122
radiation (refl1d.snsdata.Liquids attribute), 134
radiation (refl1d.snsdata.Magnetic attribute), 136
readfile() (refl1d.snsdata.ANDR method), 88
readfile() (refl1d.snsdata.MAGIK method), 89
readfile() (refl1d.snsdata.NCNRData method), 90
readfile() (refl1d.snsdata.NG1 method), 91
readfile() (refl1d.snsdata.NG7 method), 93
readfile() (refl1d.snsdata.PBR method), 95
readfile() (refl1d.snsdata.XRay method), 97
rebin2d_float32() (in module refl1d.reflmodule), 129
rebin2d_float64() (in module refl1d.reflmodule), 129
rebin2d_uint16() (in module refl1d.reflmodule), 129
rebin2d_uint32() (in module refl1d.reflmodule), 129
rebin2d_uint64() (in module refl1d.reflmodule), 129
rebin2d_uint8() (in module refl1d.reflmodule), 129
rebin_float32() (in module refl1d.reflmodule), 129
rebin_float64() (in module refl1d.reflmodule), 129
rebin_uint16() (in module refl1d.reflmodule), 130
rebin_uint32() (in module refl1d.reflmodule), 130
rebin_uint64() (in module refl1d.reflmodule), 130
rebin_uint8() (in module refl1d.reflmodule), 130
refl() (in module refl1d.abeles), 53
refl1d.abeles (module), 53
refl1d.cheby (module), 54
refl1d.dist (module), 56
refl1d.errors (module), 56
refl1d.experiment (module), 60
refl1d.fitplugin (module), 64
refl1d.freeform (module), 65
refl1d.fresnel (module), 68
refl1d.garefl (module), 68
refl1d.instrument (module), 69
refl1d.magnetism (module), 75
refl1d.material (module), 77
refl1d.materialdb (module), 80
refl1d.model (module), 81
refl1d.mono (module), 84
refl1d.names (module), 85
refl1d.ncnrdata (module), 85
refl1d.polymer (module), 98
refl1d.probe (module), 103
refl1d.profile (module), 124
refl1d.reflectivity (module), 127
refl1d.reflmodule (module), 129
refl1d.resolution (module), 130
refl1d.snsdata (module), 133
refl1d.staj (module), 137
refl1d.stajconvert (module), 144
refl1d.stitch (module), 144
refl1d.support (module), 145
refl1d.util (module), 146
reflectivity() (in module refl1d.reflectivity), 127
reflectivity() (refl1d.dist.DistributionExperiment method), 56
reflectivity() (refl1d.experiment.Experiment method), 61
reflectivity() (refl1d.experiment.ExperimentBase method), 62
reflectivity() (refl1d.experiment.MixedExperiment method), 63
reflectivity() (refl1d.fresnel.Fresnel method), 68
reflectivity_amplitude() (in module refl1d.reflectivity), 128
reload_errors() (in module refl1d.errors), 58
render() (refl1d.cheby.ChebyVF method), 55
render() (refl1d.cheby.FreeformCheby method), 55
render() (refl1d.freform.FreeformInterface01 method), 67
render() (refl1d.freform.FreeInterface method), 66
render() (refl1d.freform.FreeLayer method), 67
render() (refl1d.magnetism.FreeMagnetism method), 76
render() (refl1d.magnetism.Magnetism method), 76
render() (refl1d.magnetism.MagnetismStack method), 76
render() (refl1d.magnetism.MagnetismTwist method), 77
render() (refl1d.model.Layer method), 83
render() (refl1d.model.Repeat method), 82
render() (refl1d.model.Slab method), 82
render() (refl1d.model.Stack method), 83
render() (refl1d.mono.FreeInterface method), 84
Index
Tlo (refl1d.instrument.Pulsed attribute), 73
Tlo (refl1d.ncnrdata.ANDR attribute), 86
Tlo (refl1d.ncnrdata.MAGIK attribute), 88
Tlo (refl1d.ncnrdata.NG1 attribute), 90
Tlo (refl1d.ncnrdata.NG7 attribute), 92
Tlo (refl1d.ncnrdata.PBR attribute), 93
Tlo (refl1d.ncnrdata.XRay attribute), 96
Tlo (refl1d.snsdata.Liquids attribute), 133
Tlo (refl1d.snsdata.Magnetic attribute), 135
TOF2L() (in module refl1d.resolution), 130
TOF_range (refl1d.instrument.Pulsed attribute), 73
TOF_range (refl1d.snsdata.Liquids attribute), 133
TOF_range (refl1d.snsdata.Magnetic attribute), 135
TOF_to_data() (in module refl1d.snsdata), 137

U
unique_L (refl1d.probe.ProbeSet attribute), 116
unpolarized_magnetic() (in module refl1d.reflectivity), 129
update() (refl1d.dist.DistributionExperiment method), 57
update() (refl1d.experiment.Experiment method), 62
update() (refl1d.experiment.ExperimentBase method), 62
update() (refl1d.experiment.MixedExperiment method), 64
update_composition() (refl1d.dist.DistributionExperiment method), 57
update_composition() (refl1d.experiment.Experiment method), 62
update_composition() (refl1d.experiment.ExperimentBase method), 63
update_composition() (refl1d.experiment.MixedExperiment method), 64
write_data() (refl1d.dist.DistributionExperiment method), 57
write_data() (refl1d.experiment.Experiment method), 62
write_data() (refl1d.experiment.MixedExperiment method), 64
write_data() (refl1d.probe.NeutronProbe method), 106
write_data() (refl1d.probe.Probe method), 113
write_data() (refl1d.probe.ProbeSet method), 117
write_data() (refl1d.probe.QProbe method), 119
write_data() (refl1d.probe.XrayProbe method), 122
write_file() (in module refl1d.snsdata), 137

X
XRay (class in refl1d.ncnrdata), 95
XrayProbe (class in refl1d.probe), 120
xs (refl1d.probe.PolarizedNeutronProbe attribute), 108
xs (refl1d.probe.PolarizedQProbe attribute), 109

V
Vacuum (class in refl1d.material), 79
view (refl1d.probe.NeutronProbe attribute), 106
view (refl1d.probe.PolarizedNeutronProbe attribute), 108
view (refl1d.probe.PolarizedQProbe attribute), 109
view (refl1d.probe.Probe attribute), 113
view (refl1d.probe.ProbeSet attribute), 116
view (refl1d.probe.QProbe attribute), 119
view (refl1d.probe.XrayProbe attribute), 122
VolumeProfile (class in refl1d.polymer), 102

W
w (refl1d.profile.Microslabs attribute), 127
wavelength (refl1d.instrument.Monochromatic attribute), 73
wavelength (refl1d.instrument.Pulsed attribute), 75
wavelength (refl1d.ncnrdata.ANDR attribute), 88
wavelength (refl1d.ncnrdata.MAGIK attribute), 90
wavelength (refl1d.ncnrdata.NG1 attribute), 92
wavelength (refl1d.ncnrdata.NG7 attribute), 93
wavelength (refl1d.ncnrdata.PBR attribute), 95
wavelength (refl1d.ncnrdata.XRay attribute), 97
wavelength (refl1d.snsdata.Liquids attribute), 135
wavelength (refl1d.snsdata.Magnetic attribute), 136
wavelength (refl1d.staj.MlayerMagnetic attribute), 140
wavelength (refl1d.staj.MlayerModel attribute), 143
wavelength_dispersion (refl1d.staj.MlayerMagnetic attribute), 140
wavelength_dispersion (refl1d.staj.MlayerModel attribute), 143
Weights (class in refl1d.dist), 57
write_data() (refl1d.dist.DistributionExperiment method), 57
write_data() (refl1d.experiment.Experiment method), 62
write_data() (refl1d.experiment.MixedExperiment method), 64
write_data() (refl1d.probe.NeutronProbe method), 106
write_data() (refl1d.probe.Probe method), 113
write_data() (refl1d.probe.ProbeSet method), 117
write_data() (refl1d.probe.QProbe method), 119
write_data() (refl1d.probe.XrayProbe method), 122
write_file() (in module refl1d.snsdata), 137