1-D Reflectometry Modeling

Release 0.6.19

Paul Kienzle

July 28, 2011

CONTENTS

CHAPTER

ONE

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

Refl1D 0.6 is provided as a Windows installer or as source:

- Windows installer: refl1d-0.6-win32.exe
- Apple installer: Refl1D 0.6.dmg
- Source: refl1d-0.6.zip

Th Windows installer walks through the steps of setting the program up to run on your machine and provides the sample data to be used in the tutorial. Installers for other platforms are not yet available, and must be built from source.

1.1.1 Building from source

Building the application from source requires some preparation.

First you will need to set up your python environment. We depend on numerous external packages. The versions listed below are a snapshot of a configuration that we are using. Both older or more recent versions may work.

Our base scientific python environment contains:

- Python 2.6 (not 3.x)
- Matplotlib 1.0.0
- Numpy 1.3.0

- Scipy 0.7.0
- WxPython 2.8.10.1
- SetupTools 0.6c9
- gcc 3.4.4
- PyParsing 1.5.2
- Periodictable 1.3

To run tests you will need:

- Nose 0.11

To build the HTML documentation you will need:

- Sphinx 1.0.4
- DocUtils 0.5
- Pyments 1.0
- Jinja2 2.5.2
- MathJax 1.0.1

The PDF documentation requires a working LaTeX installation.

Platform specific details for setting up your environment are given below.

Windows

The Python(X,Y) package contains most of the pieces required to build the application. You can select "Full Install" for convenience, or you can select "Custom Install" and make sure the above packages are selected. In particular, wx is not selected by default. Be sure to select py2exe as well, since you may want to build a self contained release package.

The Python(x,y) package supplies a C/C++ compiler, but the package does not set it as the default compiler. To do this you will need to create *C:\Python26\Lib\distutils\distutils.cfg* with the following content:

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

Once python is prepared, you can install the periodic table 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". Enter the following command in the console:

```
python -m easy_install periodictable
```

This should install periodictable and pyparsing.

Next change to the directory containing the source. This will be a command like the following:

```
cd "C:\Documents and Settings\<username>\My Documents\refl1d-src"
```

Now type the command to build and install Refl1D:

```
python setup.py install
python test.py
```

Now change to your data directory:

```
cd "C:\Documents and Settings\<username>\My Documents\data"
```

To run the program use:

```
python "C:\Python26\Scripts\refl1d" -h
```

Linux

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

On ubuntu you can use apt-get to install matplotlib, numpy, scipy, wx, nose and sphinx.

From a terminal, change to the directory containing the source and type:

```
python -m easy_install periodictable
python setup.py install
python test.py
```

This should install the refl1d file somewhere on your path.

To run the program use:

```
refl1d -h
```

OS/X

Building a useful python environment on OS/X is somewhat involved, and this documentation will be expanded to provide more detail.

You will need to download python, numpy, scipy, wx and matplotlib packages from their respective sites (use the links above). Setuptools will need to be installed by hand.

From a terminal, change to the directory containing the source and type:

```
python -m easy_install periodictable nose sphinx
python setup.py install
python test.py
```

This should install the refl1d file somewhere on your path.

To run the program use:

```
refl1d -h
```

1.1.2 Building Documentation

Building the package documentation requires a working Sphinx installation, a working LaTex installation and a copy of MathJax. Download and unzip the MathJax package into the doc/sphinx directory to install MathJax. You can then build the documentation as follows:

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

Note that this only works under cygwin/msys for now since we are using *make*. There is a skeleton *make.bat* in the directory that will work using *cmd* but it doesn't yet build PDF files.

You can see the result by pointing your browser to:

```
refl1d/doc/_build/html/index.html
refl1d/doc/_build/latex/Refl1D.pdf
```

As of this writing, the \AA LaTeX command for the Angstrom symbol is not available in the MathJax distribution. We patched jax/input/TeX/jax.js with the additional symbol AA using:

```
  // Ord symbols
  S:            '00A7',
+ AA:           '212B',
  aleph:        ['2135',{mathvariant: MML.VARIANT.NORMAL}],
```

If you are using unusual math characters, you may need similar patches for your own documentation.

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/cm^3| 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 sphinx versions (1.0.7 as of this writing) 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.

1.1.3 Windows Installer

To build a windows standalone executable with py2exe you may first need to create an empty file named *C:\Python26\Lib\numpy\distutils\tests\__init__.py*. 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:

- Visual C++ 2008 Redistributable Package (x86) 11/29/2007
- 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.

1.2 Server installation

Refl-1D jobs can be submitted to a remote batch queue for processing. This allows users to share large clusters for faster processing of the data. The queue consists of several components.

- job controller

  http service layer which allows users to submit jobs and view results

- queue

  cluster management layer which distributes jobs to the working nodes

- worker

  process monitor which runs a job on the working nodes

- mapper

  mechanism for evaluating R(x_i) for different x_i on separate CPUs

If you are setting up a local cluster for performing reflectometry fits, then you will need to read this section, otherwise you can continue to the next section.

Assuming that the refl1d server is installed as user reflectometry in a virtualenv of ~/reflserv, MPLCONFIGDIR is set to ~/reflserve/.matplotlib, and reflworkd has been configured, you can start with the following profile:

1.2.1 Job Controller

`jobqueue` is an independent package within refl1d. It implements an http API for interacting with jobs.

It is implemented as a WSGI python application using Flask

<VirtualHost *:80> ServerAdmin pkienzle@nist.gov ServerName www.reflectometry.org ServerAlias reflectometry.org ErrorLog logs/reflectometry-error_log CustomLog logs/reflectometry-access_log common

WSGIDaemonProcess reflserve user=pkienzle group=refl threads=3 WSGIScriptAlias /queue /home/pkienzle/reflserve/www/jobqueue.wsgi

<Directory "/home/pkienzle/reflserve/www"> WSGIProcessGroup reflserve WSGIApplicationGroup %{GLOBAL} Order deny,allow Allow from all

</Directory>

DocumentRoot /var/www/reflectometry <Directory "/var/www/reflectometry/">

AllowOverride All

</Directory>

</VirtualHost>

There is a choice of two different queuing systems to configure. If your environment supports a traditional batch queue you can use it to manage cluster resources. New jobs are added to the queue, and when they are complete, they leave their results in the job results directory. Currently only slurm is supported, but supporting torque as well would only require a few changes.

You can also set up a central dispatcher. In that case, you will have remote clusters pull jobs from the server when they are available, and post the results to the job results directory when they are complete. The remote cluster may be set up with its own queuing system such as slurm, only taking a few jobs at a time from the dispatcher so that other clusters can share the load.

1.2.2 Cluster

If you are using the dispatcher queuing system, you will need to set up a work daemon on your cluster to pull jobs from the queue. This requires adding reflworkerd to your OS initialization scripts.

1.2.3 Security

Because the jobqueue can run without authentication we need to be especially concerned about the security of our system. Techniques such as AppArmor or virtual machines with memory mapped file systems provide a relatively safe environment to support anonymous computing.

Note: It easy to add authentication to flask, but we can avoid it if our community plays nice — every beamline should supply sufficient compute power to host their user base, either directly or through one of the many cloud computing services.

To successfully set up AppArmor, there are a few operations you need.

Each protected application needs a profile, usually stored in /etc/apparmor.d/path.to.application. With the reflenv virtural environment in the reflectometry user, the following profile would be appropriate for the worker daemon:

```
-- /etc/apparmor.d/home.reflectometry.reflenv.bin.reflworkd
#include <tunables/global>

/home/reflectometry/reflenv/bin/reflworkd {
 #include <abstractions/base>
 #include <abstractions/python>

 /bin/dash cx,
 /home/reflectometry/reflenv/bin/python cx,
 /home/reflectometry/reflenv/** r,
 /home/reflectometry/reflenv/**.{so,pyd} mr,
 /home/reflectometry/.reflserve/.matplotlib/* rw,
 /home/reflectometry/.reflserve/worker/** rw,
}
```

This gives read access/execute access to python and its C extensions, and read access to everything else in the virtual environment.

The rw access to .reflserve is potentially problematic. Hostile models can interfere with each other if they are running at the same time. In particular, they can inject html into the returned data set which can effectively steal authentication credentials from other users through cross site scripting attacks, and so would not be appropriate on a closed server.

Restricting the model to .reflserve/worker/jobid/** would reduce this risk, but this author does not know how to do so without elevating reflworkd privileges to root.

A similar profile could be created for the job server, and indeed, any web service you have on your machine, but this is less critical since it is not running user models.

Once the profile is in place, restart the apparmor.d daemon to enable it:

```
sudo service apparmor restart
```

You can debug the profile by running a trace while the program runs unrestricted. To start the trace, use:

```
sudo genprof /path/to/application
```

Switch to another window then run:

```
/path/to/app
```

When your application is complete, return to the genprof window and hit 'S' to scan /var/log/syslog for file and network access. Follow the prompts to update the profile. The documentation on AppArmor on Ubuntu and AppArmor on SUSE is very helpful here.

To reload a profile after running the trace, use:

```
sudo apparmor_parser -r /etc/apparmor.d/path.to.application
```

To delete a profile that you no longer need:

```
sudo rm /etc/apparmor.d/path.to.application
sudo service apparmor restart
```

1.3 Contributing Changes

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:

> http://github.com/reflectometry/refl1d

You can obtain a copy via git using:

```
git clone https://github.com/reflectometry/refl1d.git
    cd refl1d
    python setup.py develop
```

By using the *develop* keyword on setup.py, changes to the files in the package are immediately available without the need to run setup.py install each time.

Track updates to the original package using:

```
git pull
```

If you find you need to modify the package, 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. Using the the nose test package, you can run both sets of tests:

```
easy_install nose
python2.5 tests.py
python2.6 tests.py
```

When all the tests run, generate a patch and send it to the DANSE Project mailing list at danse-dev@cacr.caltech.edu:

```
git diff > patch
```

Alternatively, create a project fork at github and we can pull your changes directly from your repository.

Windows user can use TortoiseGit package which provides similar operations.

Building the package documentation requires a working sphinx installation and in addition, a copy of MathJax to view the equations. Download and unzip the MathJax package into the doc/sphinx directory to install MathJax. You can then build the documentation as follows:

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

You can see the result by pointing your browser to:

```
refl1d/doc/_build/html/index.html
refl1d/doc/_build/latex/PeriodicTable.pdf
```

As of this writing, the \AA LaTeX command for the Angstrom symbol is not available in the MathJax distribution. We patched jax/input/TeX/jax.js with the additional symbol AA using:

```
  // Ord symbols
  S:            '00A7',
+ AA:           '212B',
  aleph:        ['2135',{mathvariant: MML.VARIANT.NORMAL}],
```

If you are using unusual math characters, you may need similar patches for your own documentation.

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/cm^3| to hide the details. The complete list of macros is available in

> doc/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/conf.py.

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:

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 is actively developed under DANSE project and is maintained by the DANSE project.

Project developers include:

```
Paul Kienzle
James Krycka
Nikunj Patel
Christopher Metting
Ismet Sahin
Ziwen Fu
Wenwu Chen
Alexander Mont
David Tighe
```

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.

CHAPTER

TWO

TUTORIAL

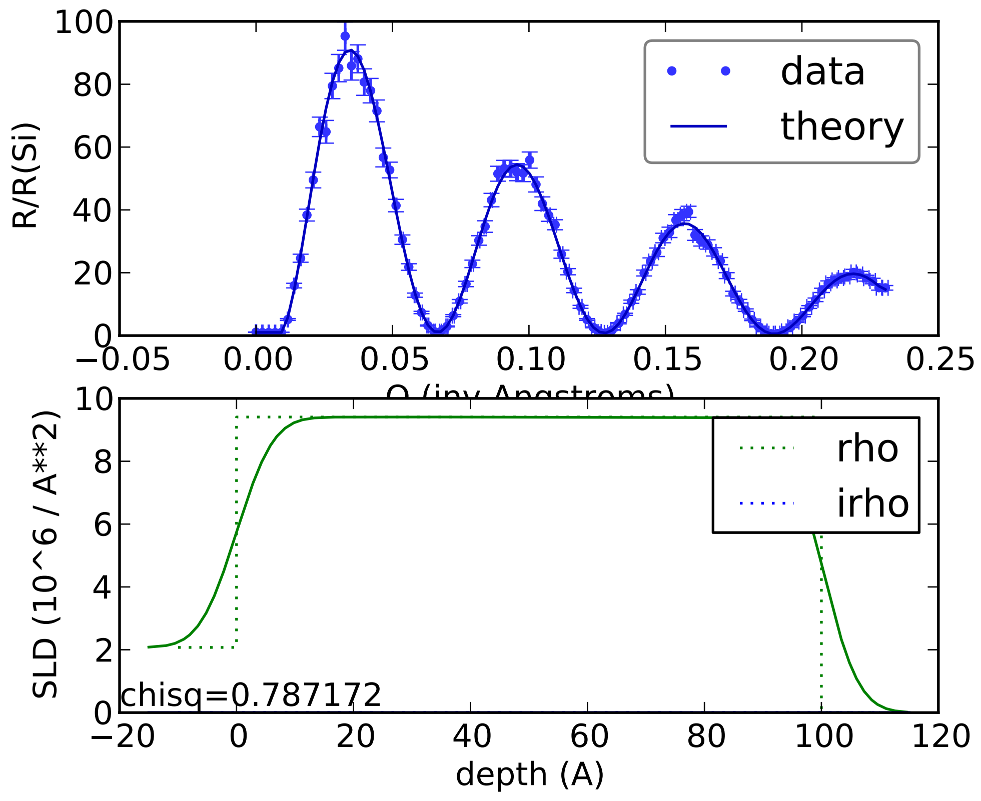
This tutorial will describe walk through the steps of setting up a model with Python scripting. Scripting allows the user to create complex models with many constraints relatively easily.

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 $\exp(-2k_n k_{n+1}\sigma^2)$ interface factored in.

This model is defined in `nifilm.py`.

You can preview the model on the command line:

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

Lets 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:

`::`

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:

`::`

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.

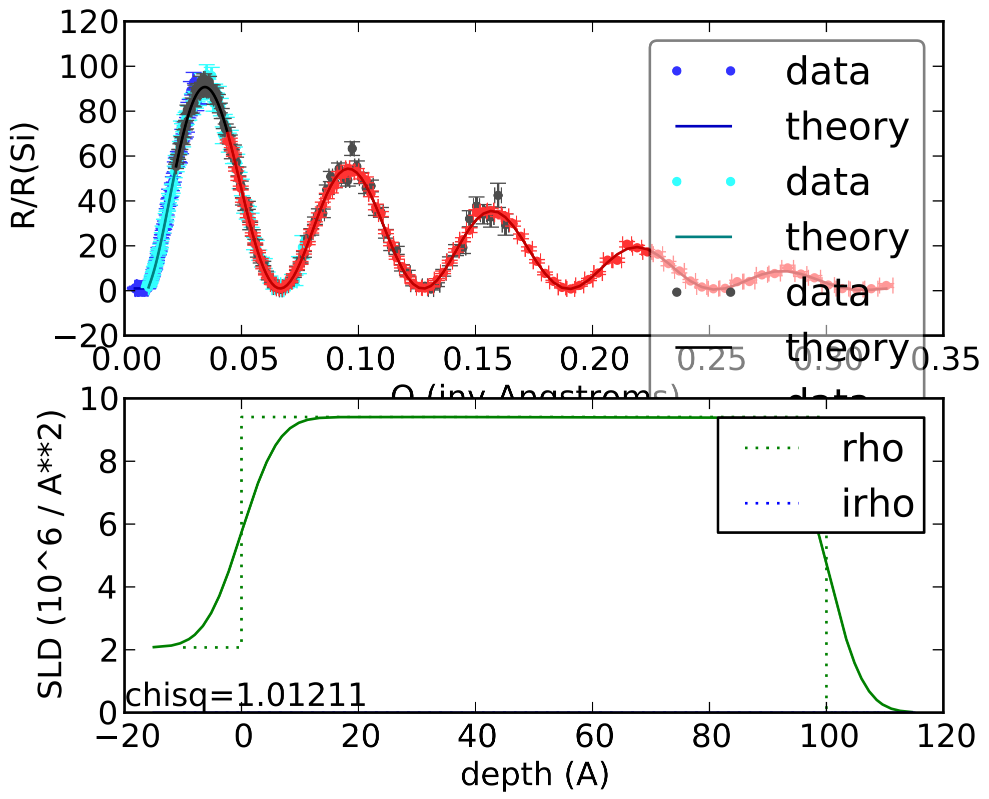
```
M.simulate_data(5)
```

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:



This model is defined in `nifilm-tof.py`

The sample definition is the same:

```
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.

```
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.

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

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:

```
from refl1d.names import *

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

instrument = SNS.Liquids()
```

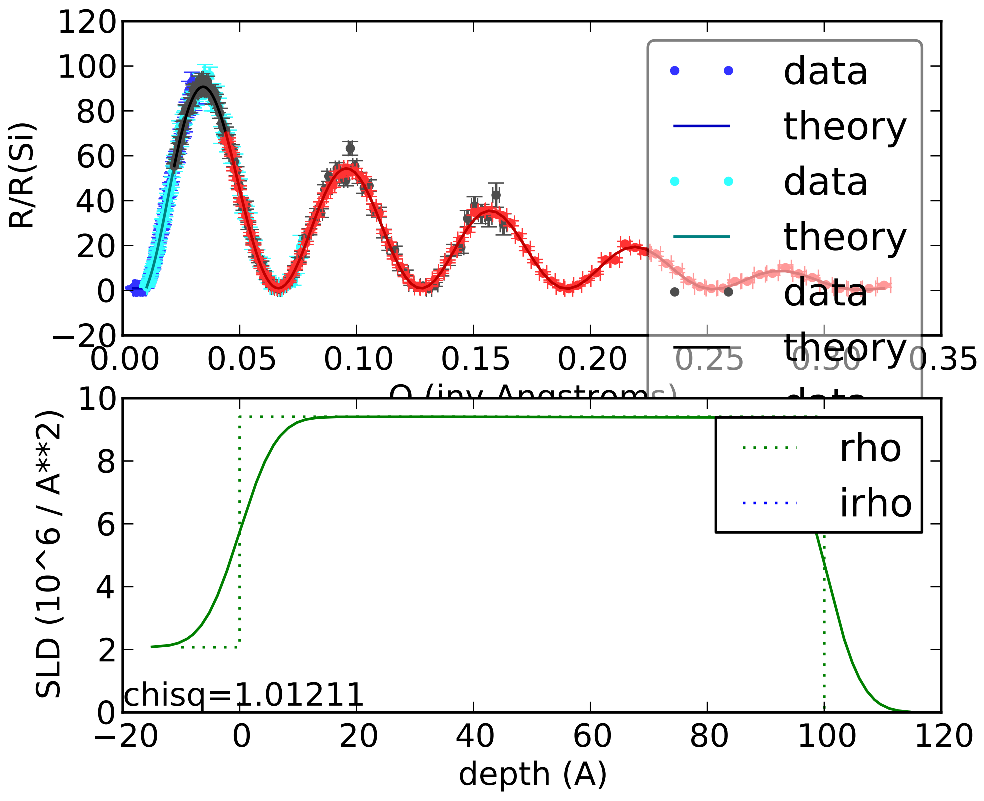
In this case we are loading multiple data sets into the same `ProbeSet` object. If your reduction program stitches together the data for you, then you can simply use `probe=instrument.load('file')`.

```
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.

```
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:

```
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.

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

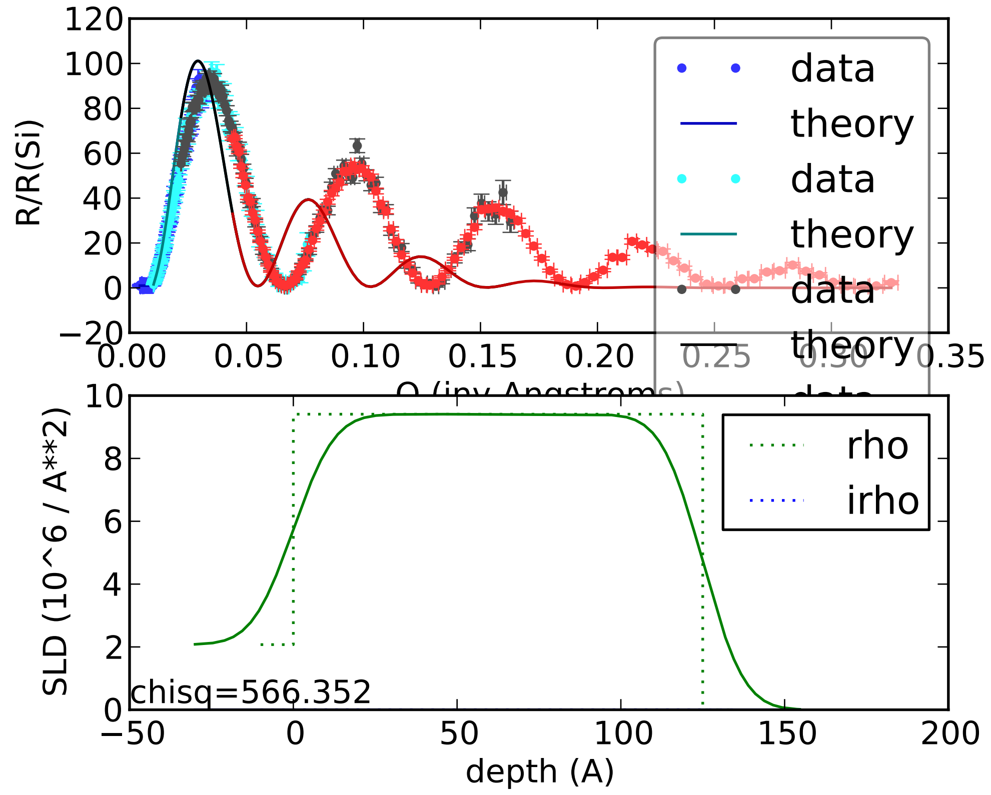
We are also going to let the interfacial roughness between the layers vary. The interface between two layers is defined by the width of the interface on top of the layer below. Here we are restricting the silicon:nickel interface to the interval $[3, 12]$ and the nickel:air interface to the range $[0,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

```
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:

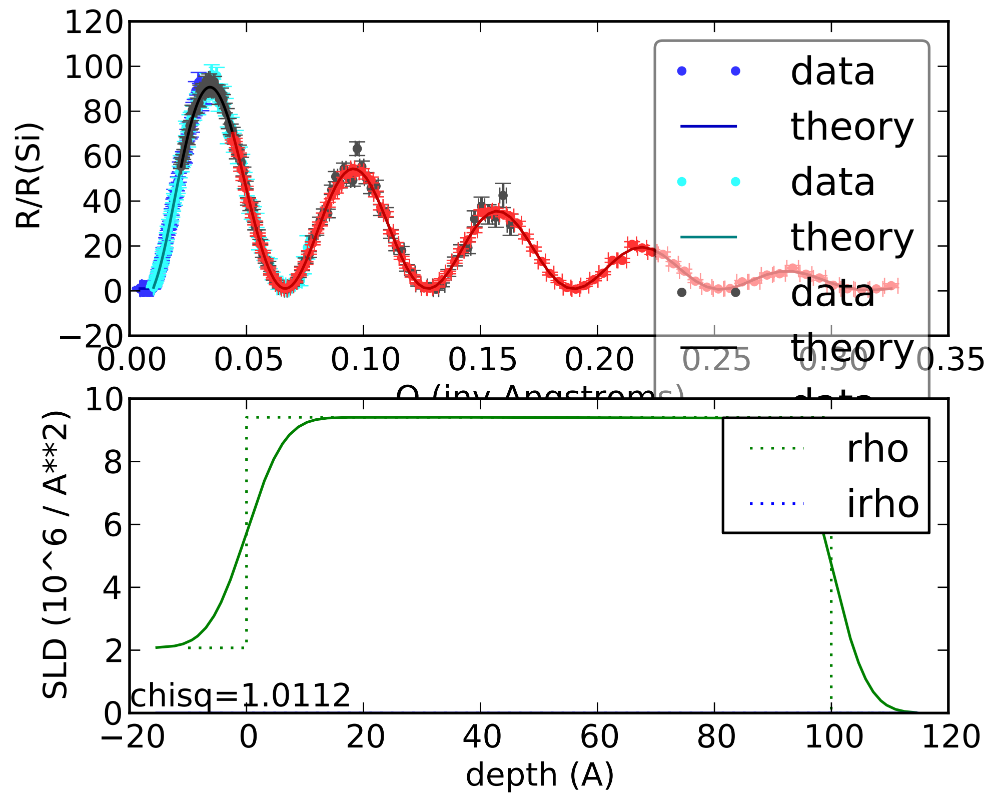


We can now load and run the fit:

```
# 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: χ^2 will be approximately 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.

```
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.

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

The remainder of the model definition is unchanged.

```
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

```
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 ρ. Note that we are using the imaginary ρ_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 `PolymerBrush` layer, which understands that the system is compose of polymer plus solvent, and that the polymer chains tail off like:

$$V(z) = \begin{cases} V_o & \text{if } z <= z_o \\ V_o(1 - ((z - z_o)/L)^2)^p & \text{if } z_o < z < z_o + L \\ 0 & \text{if } z >= z_o + L \end{cases}$$

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

```
# === 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

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

The stack notation `material(thickness, interface) | ...` is performing a number of tasks for you. One thing it is doing is wrapping materials (which are objects that understand scattering length densities) into slabs (which are objects that understand thickness and interface). These slabs are then gathered together into a stack:

```
L_silicon = Slab(material=silicon, thickness=0, interface=5)
L_SiOx = Slab(material=SiOx, thickness=100, interface=5)
L_D_initiator = Slab(material=D_initiator, thickness=100, interface=20)
L_D_brush = copy(D_brush)
L_D_brush.thickness = Parameter.default(400,name=D_brush.name+" thickness")
L_D_brush.interface = Parameter.default(0,name=D_brush.name+" interface")
L_D_toluene = Slab(material=D_toluene)
D = Stack([L_silicon, L_SiOx, L_D_initiator, L_D_brush, L_D_toluene])
```

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

```
# 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*

```
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 χ^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

```
# === 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:

```
# === 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)
```

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 steps when generating the tethered polymer interface. The dA parameter allows these steps to be joined together into larger slabs, with each slab having $(\rho_{\max} - \rho_{\min})w < \Delta A$.

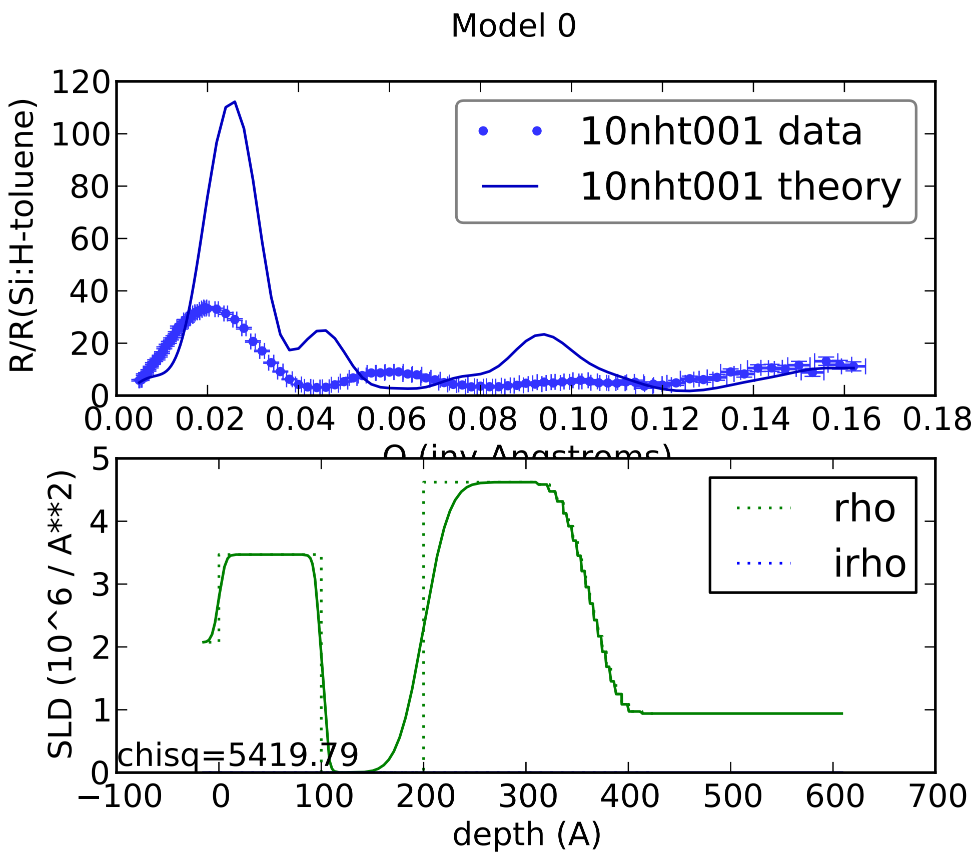
```
# === Problem definition ===
D_model = Experiment(sample=D, probe=D_probe, dz=0.5, dA=1)
H_model = Experiment(sample=H, probe=H_probe, dz=0.5, dA=1)
models = H_model, D_model
```

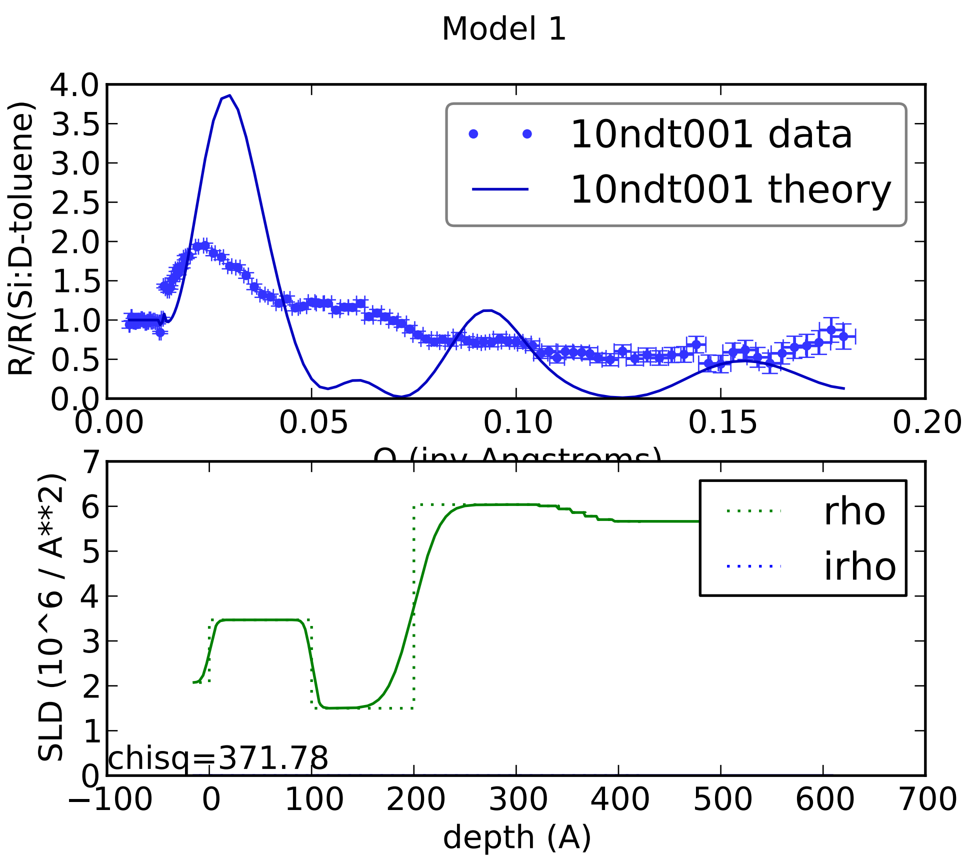
This is a multifit problem where both models contribute to the goodness of fit measure χ^2. Since no weight vector was defined the fits have equal weight.

```
problem = MultiFitProblem(models=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 ρ_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:


Model 0
R/R(Si:H-toluene)
10nht001 data
10nht001 theory
Q (inv Angstroms)
SLD (10^6 / A**2)
rho
irho
chisq=5419.79
depth (A)



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

```
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)

# === 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 = MultiFitProblem(models=models)
problem.name = "tethered"
```

The model can be fit using the parallel tempering optimizer:

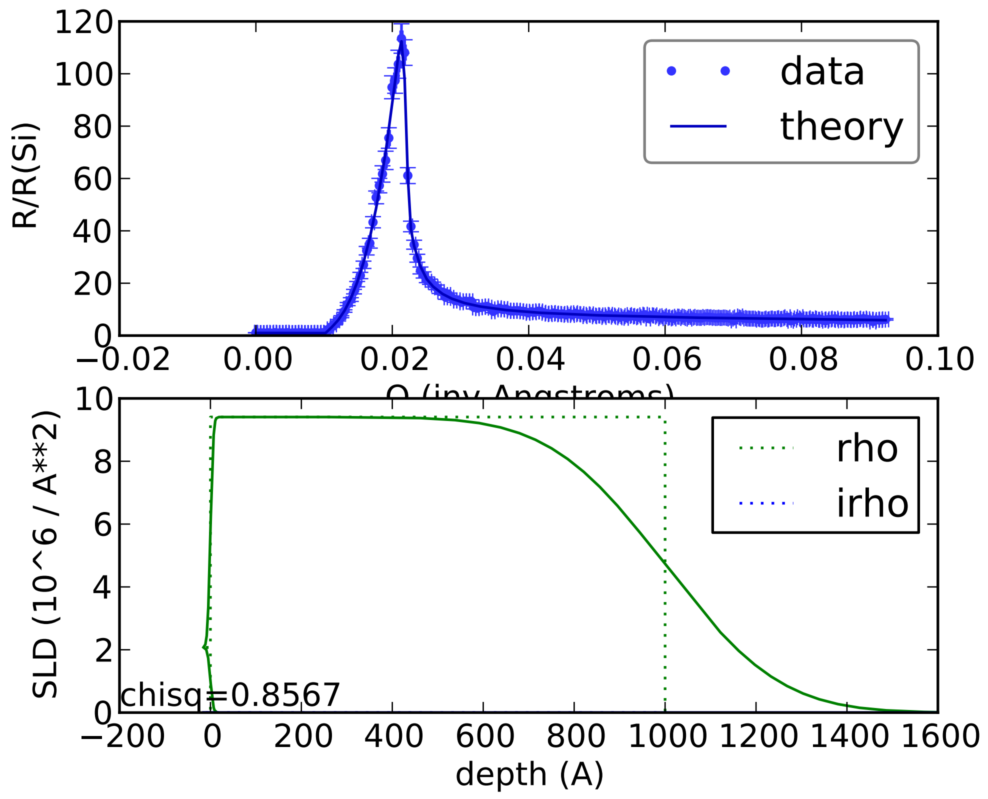
```
$ refl1d tethered.py --fit=pt --store=T1
```

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.

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

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

```
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.

```
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.

```
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.

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

We will want to fit the thicknesses and interfaces as usual.

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

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`:

```
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.

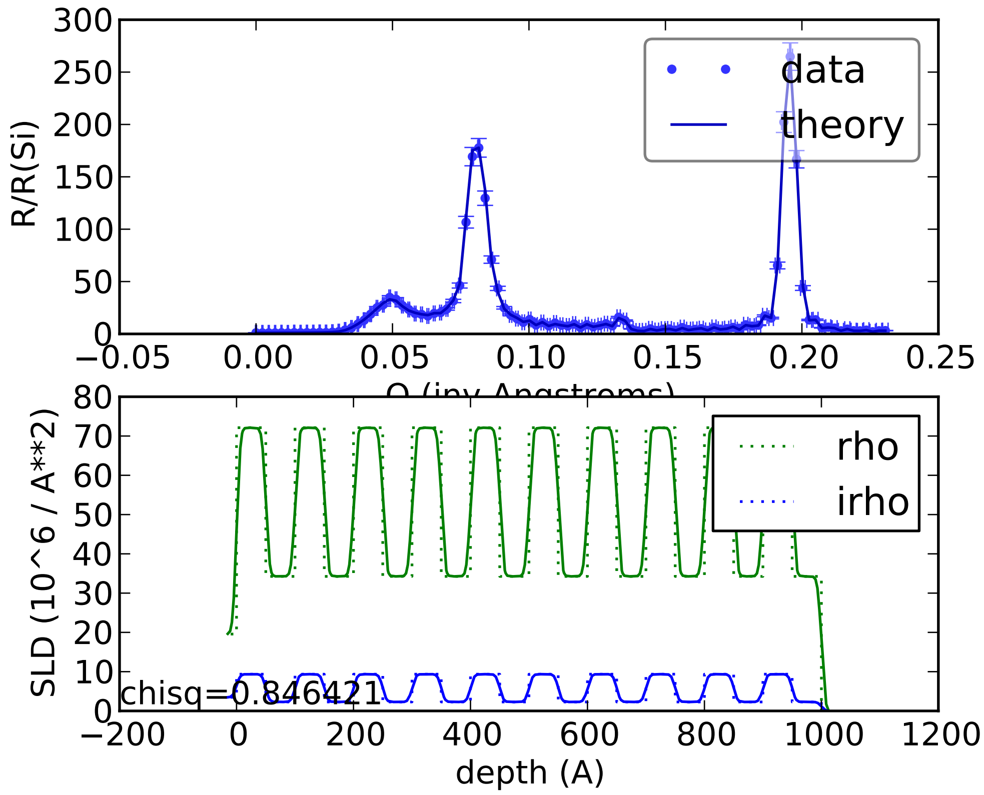
2.4.1 Hard material structures

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

Singh, S., Basu, S., Bhatt, P., Poswal, A.K., Phys. Rev. B, 79, 195435 (2009)

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

```
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.

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

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

```
# 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.

```
bilayer[0].interface.range(0,30)
bilayer[1].interface.range(0,30)
sample[0].interface.range(0,30)
sample[1].interface.range(0,30)
```

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:

```
# 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

```
# 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.

```
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.

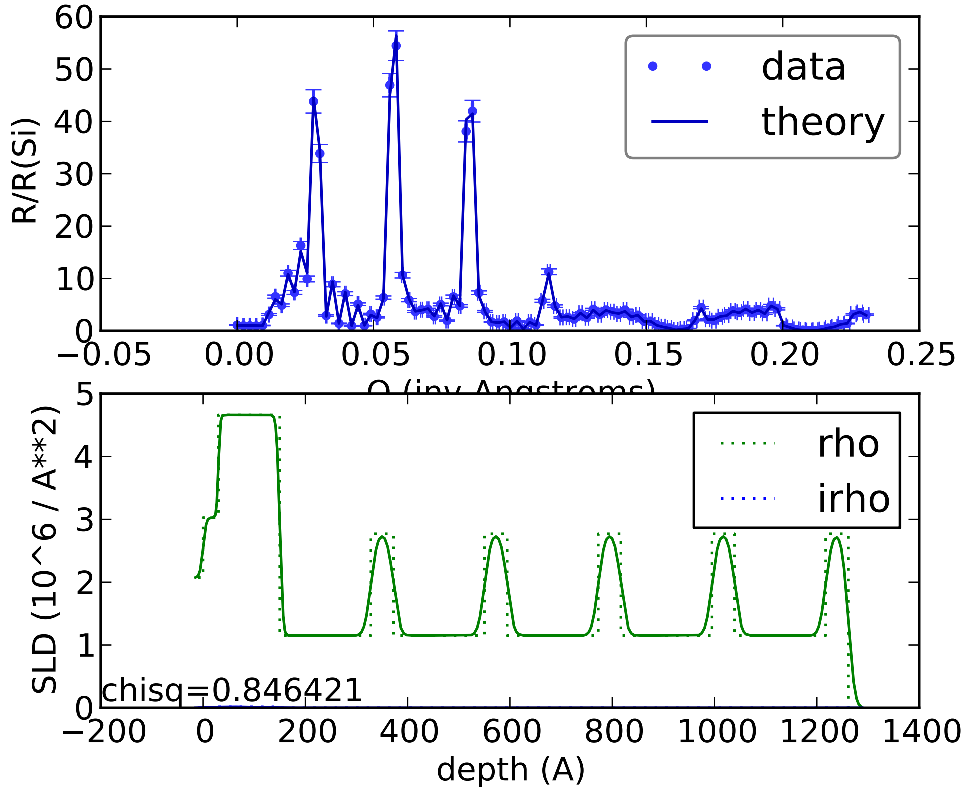
```
T = numpy.linspace(0, 5, 100)
probe = XrayProbe(T=T, dT=0.01, L=4.75, dL=0.0475)
M = Experiment(probe=probe, sample=sample)
M.simulate_data(5)
problem = FitProblem(M)
```

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`.



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

```
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.

```
# == 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.

```
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).

```
bilayer = PDADMA_PSS(178,10) | PDADMA_dPSS(44.3,10)
```

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.

```
sample = (silicon(0,5) | chrome(30,3) | gold(120,5)
          | (bilayer)*4 | PDADMA_PSS(178,10) | PDADMA_dPSS(44.3,10) | air)
```

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.

```
# == Fit parameters ==
PDADMA_dPSS.rho.range(1.15,2.77)
PDADMA_PSS.rho.range(1.15,2.77)
```

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.

```
sample[3][0].interface.range(5,45)
sample[3][1].interface.range(5,45)
```

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

```
sample[3].interface.range(5,45)
```

Because the top bilayer has different dynamics, we optimize the interfaces independendly. Although we want the optimiser to threat 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.

```
sample[4].interface.range(5,45)
sample[5].interface.range(5,45)
```

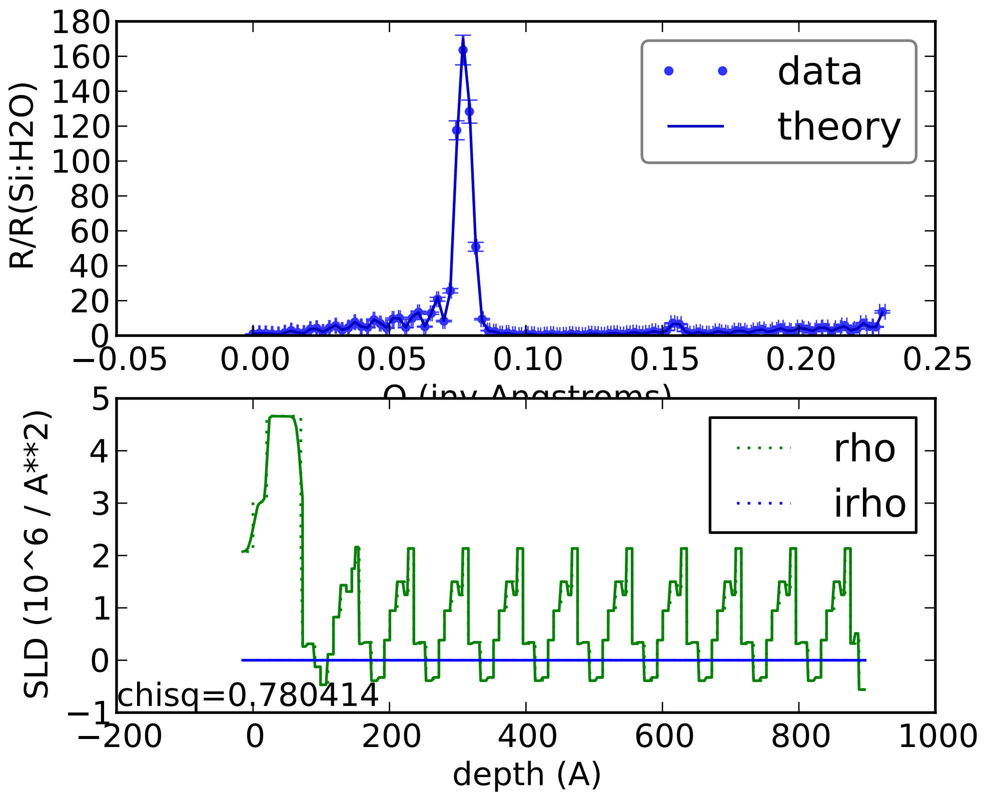
Finally we need to associate the sample with a measurement. We do not have the measurements from the paper available, so instead we will simulate a measurement but setting up a neutron probe whose incident angles range from 0 to 5 degrees in 100 steps. The simulated measurement is returned together with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

```
# == Data ==
T = numpy.linspace(0, 5, 100)
probe = NeutronProbe(T=T, dT=0.01, L=4.75, dL=0.0475)

M = Experiment(probe=probe, sample=sample)
M.simulate_data(5)

problem = FitProblem(M)
```

2.4.3 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 straight forward:

```
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

```
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 *n1* and *n3* small, but make *n2* large enough to define an interesting repeat structure.

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

Free layers have a thickness, horizontal control points z varying in $[0,1]$, real and complex SLD ρ and ρ_i, and the material above and below.

```
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.

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

Finally, simulate the resulting model.

```
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)
```

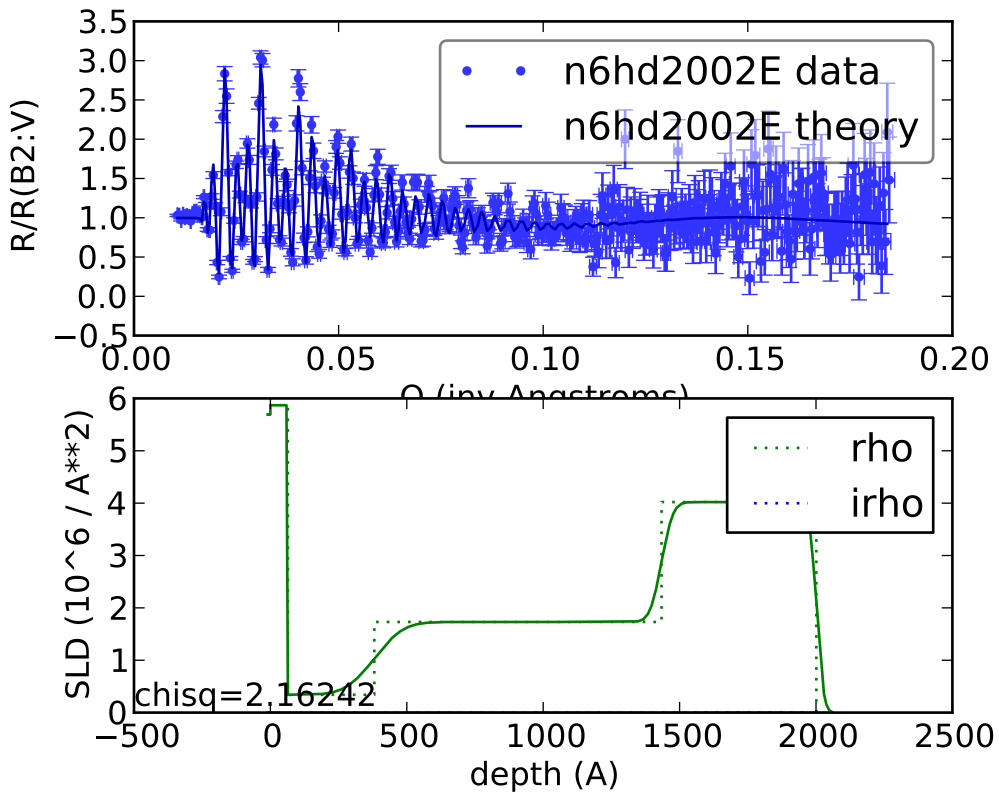
2.5 MLayer Models

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:

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

This shows the model plot:



and the available model parameters:

```
.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')
      .thickness = Parameter(316.991, name='B1 thickness')
    [3]
      .interface = Parameter(33.2095, name='M2 interface')
      .material
        .irho = Parameter(6.93684e-05, name='M2 irho')
        .rho = Parameter(1.73106, name='M2 rho')
      .thickness = Parameter(1052.77, name='M2 thickness')
    [4]
      .interface = Parameter(20.6753, name='M1 interface')
      .material
        .irho = Parameter(0.00137419, name='M1 irho')
        .rho = Parameter(4.02059, name='M1 rho')
      .thickness = Parameter(567.547, name='M1 thickness')
    [5]
      .interface = Parameter(4.24661e-11, name='V interface')
      .material
        .irho = Parameter(0, name='V irho')
        .rho = Parameter(0, name='V rho')
      .thickness = Parameter(0, name='V thickness')
  .thickness = stack thickness:2091.32

[chisq=2.16242, nllf=408.697]
```

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
M = load_mlayer("De2_VATR.staj")

# 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])
```

CHAPTER

THREE

USER'S GUIDE

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 reflectivty.

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.

Introduction

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

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.

Samples

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`.

Experiments

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:

```
>>> 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:

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

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:

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

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 (θ_1, λ_1) and (θ_2, λ_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 vaues 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, which also acts to spread the beam. The degree of spread can be estimated from the full-width at half max (FWHM) of a rocking curve at known slit settings. The expected FWHM will be $\frac{1}{2}(s_1 + s_2)/(d_1 - d_2)$. The difference between this and the measured FWHM is the sample_broadening value. A second order effect is that at low angles the warping will cast shadows, changing the resolution and intensity in very complex ways.

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 σ_λ and angle σ_θ using propagation of errors:

$$\begin{aligned}
\sigma_Q^2 &= \left|\frac{\partial Q}{\partial \lambda}\right|^2 \sigma_\lambda^2 + \left|\frac{\partial Q}{\partial \theta}\right|^2 \sigma_\theta^2 + 2\left|\frac{\partial Q}{\partial \lambda}\frac{\partial Q}{\partial \theta}\right|^2 \sigma_{\lambda\theta} \\
Q &= 4\pi\sin(\theta)/\lambda \\
\frac{\partial Q}{\partial \lambda} &= -4\pi\sin(\theta)/\lambda^2 = -Q/\lambda \\
\frac{\partial Q}{\partial \theta} &= 4\pi\cos(\theta)/\lambda = \cos(\theta)\cdot Q/\sin(\theta) = Q/\tan(\theta)
\end{aligned}$$

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 θ, 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-σ 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 θ_o at the start of the opening slits region you can compute $\Delta\theta/\theta_o$, and later scale that to your particular θ:

$$\Delta\theta(Q) = \Delta\theta/\theta_o \cdot \theta(Q)$$

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 ΔQ 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 refectivity $R(Q_i)$ computed at points $Q_i \in Q_\text{calc}$

$$\begin{aligned}\bar{R}_i(q) &= m_i q + b_i \\ m_i &= (R_{i+1} - R_i)/(Q_{i+1} - Q_i) \\ b_i &= R_i - m_i Q_i\end{aligned}$$

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_min to i_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_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_2O, 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 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 a 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 a 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.[1] 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:

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

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 is 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, ρ_i, $\rho_{\rm inc}$) 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 $\rho_{\rm inc}$ 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`

[1] M.R. Daymond, P.J. Withers and M.W. Johnson; The expected uncertainty of diffraction-peak location”, Appl. Phys. A 74 [Suppl.], S112 - S114 (2002). http://dx.doi.org/10.1007/s003390201392

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

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:

```
>> 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:

```
>> 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:

```
>> 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 (`refl1d.mono`)
- parameteric B-splines (`refl1d.freeform`)
- '**Chebyshev interpolating polynomials** <http://en.wikipedia.org/wiki/Chebyshev_polynomials>'_
  (`refl1d.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.

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 an function over an interval with respect to the maximum norm. The interpolating polynomial is a weighted sum $\sigma_{i=0}^{n} c_i T_i(z)$ 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

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 the **'new_layers'_** section: 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.

```
>>> 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))
>>> r = reflamp(kz=Qin/2, depth=w, rho=rho, irho=irho, sigma=sigma)
>>> 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: %.2g  R: %.5g"%(Qi,Ri) for Qi,Ri in zip(Q,R))
```

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 --parellel 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.

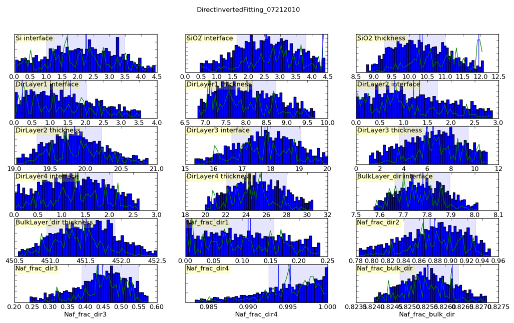
DREAM can be started with a variety of initial populations. The random population (init=random) distributes the initial points using a uniform distribution across the space of the parameters. Latin hypersquares (init=lhs) improves on random by making sure that there is on value for each subrange of every 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.

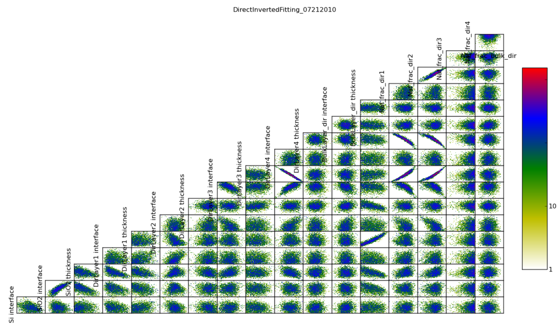
A common command line for running DREAM is:

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

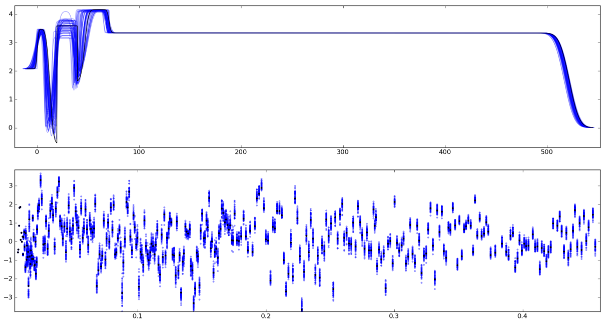
Bayesian uncertainty analysis us described in the GUM Supplement 1,[8] and is a valid technique for reporting parameter uncertainties in NIST publications. Given sufficient burn time, points in the search space will be visited with probability proportional to the goodness of fit. The file T1/model.err 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.

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]:

```
>>> from dream import views
>>> selection={2: (0.8,1.0), 4:(0.2,0.4),...}
>>> views.plot_vars(state, selection=selection)
>>> views.plot_corrmatrix(state, selection=selection)
```

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:

```
>>> state.derive_vars(lambda p: p[0]+p[1], labels=["x+y"])
```

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

```
>>> state.derive_vars(lambda p: (p[0]*p[1],p[0]-p[1]), labels=["x*y","x-y"])
```

These new parameters will show up in your plots:

```
>>> 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:

```
# ... 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 http://github.com/reflecometry/refl1d.

3.7.4 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 agressively 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 --sto
```

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.

3.7.5 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
```

3.7.6 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.

3.7.7 References

WH Press, BP Flannery, SA Teukolsky and WT Vetterling, Numerical Recipes in C, Cambridge University Press

I. Sahin (2011) Random Lines: A Novel Population Set-Based Evolutionary Global Optimization Algorithm. Lecture Notes in Computer Science, 2011, Volume 6621/2011, 97-107 DOI:10.1007/978-3-642-20407-4_9

Vrugt, J. A., ter Braak, C. J. F., Diks, C. G. H., Higdon, D., Robinson, B. A., and Hyman, J. M.:Accelerating Markov chain Monte Carlo simulation by differential evolution with self-adaptive randomized subspace sampling, Int. J. Nonlin. Sci. Num., 10, 271–288, 2009.

Kennedy, J.; Eberhart, R. (1995). "Particle Swarm Optimization". Proceedings of IEEE International Conference on Neural Networks. IV. pp. 1942–1948. doi:10.1109/ICNN.1995.488968

23. Huyer and A. Neumaier, Snobfit - Stable Noisy Optimization by Branch and Fit, ACM Trans. Math. Software 35 (2008), Article 9.

Storn, R.: System Design by Constraint Adaptation and Differential Evolution, Technical Report TR-96-039, International Computer Science Institute (November 1996)

Swendsen RH and Wang JS (1986) Replica Monte Carlo simulation of spin glasses Physical Review Letters 57 : 2607-2609

BIPM, IEC, IFCC, ILAC, ISO, IUPAC, IUPAP, and OIML. Evaluation of measurement data – Supplement 1 to the 'Guide to the expression of uncertainty in measurement' – Propagation of distributions using a Monte Carlo method. Joint Committee for Guides in Metrology, JCGM 101 <http://www.bipm.org/utils/common/documents/jcgm/JCGM_101_2008_E.pdf>, 2008.

CHAPTER

FOUR

REFERENCE

4.1 refl1d.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] | inv angstrom] Scattering vector $2 * \pi * \sin(heta)/\lambda$. This is $Q_z/2$.

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 refl1d.bspline - B-Spline interpolation library

`bspline`	Evaluate the B-spline at positions xt in [0,1].
`bspline_control`	
`demo`	
`demo_interp`	
`max`	
`min`	
`pbs`	Evaluate the parametric B-spline x(t),y(t) in [0,1].
`pbs_control`	
`speed_check`	
`test`	

BSpline calculator.

Given a set of knots, compute the degree 3 B-spline and any derivatives that are required.

`refl1d.bspline.`**`bspline`**(*y*, *xt*, *clamp=True*)

Evaluate the B-spline at positions xt in [0,1].

The knots are assumed to be equally spaced within 0,1.

The spline goes through the control points at the ends. If clamp is True, the derivative of the spline at both ends is zero. If clamp is False, the derivative at the ends is equal to the slope connecting the final pair of control points.

`refl1d.bspline.`**`bspline_control`**(*y*, *clamp=True*)

`refl1d.bspline.`**`demo`**()

`refl1d.bspline.`**`demo_interp`**()

`refl1d.bspline.`**`max`**(*a*, *b*)

`refl1d.bspline.`**`min`**(*a*, *b*)

`refl1d.bspline.`**`pbs`**(*x*, *y*, *t*, *clamp=True*, *parametric=False*)

Evaluate the parametric B-spline x(t),y(t) in [0,1].

The knots are assumed to be equally spaced within 0,1. x values are sorted.

The spline goes through the control points at the ends. If clamp is True, the derivative of the spline at both ends is zero. If clamp is False, the derivative at the ends is equal to the slope connecting the final pair of control points.

If parametric is False, then parametric points t' are chosen such that x(t') = t. The control points x must be linearly increasing for this to work.

`refl1d.bspline.`**`pbs_control`**(*x*, *y*, *clamp=True*)

`refl1d.bspline.`**`speed_check`**()

`refl1d.bspline.`**`test`**()

4.3 refl1d.cheby - Freeform - Chebyshev

`ChebyVF`	Material in a solvent
`FreeformCheby`	A freeform section of the sample modeled with Chebyshev polynomials.
`cheby_approx`	Return the coefficients for the order n chebyshev approximation to
`cheby_coeff`	Compute chebyshev coefficients for a polynomial of order n given the function evaluated at the chebyshev points for order n.
`cheby_points`	Return the points in at which a function must be evaluated to
`cheby_val`	Evaluate the chebyshev approximation c at points x.

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:

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

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 * profile(z) + solvent.sld * (1 - profile(z))
```

constraints()
Constraints

find(*z*)
Find the layer at depth z.

Returns layer, start, end

parameters()

render(*probe*, *slabs*)

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

parameters()

render(*probe*, *slabs*)

refl1d.cheby.**cheby_approx**(*n, f, range=[0, 1]*)
Return the coefficients for the order n chebyshev approximation to function f evaluated over the range [low,high].

refl1d.cheby.**cheby_coeff**(*fx*)
Compute chebyshev coefficients for a polynomial of order n given the function evaluated at the chebyshev points for order n.

This can be used as the basis of a direct interpolation method where the n control points are positioned at cheby_points(n).

refl1d.cheby.**cheby_points**(*n, range=[0, 1]*)
Return the points in at which a function must be evaluated to generate the order n Chebyshev approximation function.

Over the range [-1,1], the points are $p_k = \cos(\pi(2k+1)/(2n))$. Adjusting the range to $[x_L, x_R]$, the points become $x_k = \frac{1}{2}(p_k - x_L + 1)/(x_R - x_L)$.

refl1d.cheby.**cheby_val**(*c*, *x*, *method='direct'*)
Evaluate the chebyshev approximation c at points x.

The values c_i are the coefficients for the chebyshev polynomials T_i yielding $p(x) = \sum_i c_i T_i(x)$.

4.4 refl1d.cli - Command line interface

`ParseOpts`	
`Refl1dOpts`	
`beep`	Audio signal that fit is complete.
`config_matplotlib`	Setup matplotlib.
`getopts`	
`initial_model`	
`load_problem`	
`main`	
`make_store`	
`mesh`	
`preview`	
`recall_best`	
`remember_best`	
`resynth`	
`run_profile`	Model execution time profiler.
`start_remote_fit`	Queue remote fit.
`store_overwrite_query`	
`store_overwrite_query_gui`	

class `refl1d.cli.`**`ParseOpts`**(*args*)

class `refl1d.cli.`**`Refl1dOpts`**(*args*)

Bases: `refl1d.cli.ParseOpts`

`fit`

`plot`

`transport`

`refl1d.cli.`**`beep`**()

Audio signal that fit is complete.

`refl1d.cli.`**`config_matplotlib`**(*backend*)

Setup matplotlib.

The backend should be WXAgg for interactive use, or 'Agg' for batch.

This must be called before any imports to pylab. We've done this by making sure that pylab is never (rarely?) imported at the top level of a module, and only in the functions that call it: if you are concerned about speed, then you shouldn't be using pylab :-)

`refl1d.cli.`**`getopts`**()

`refl1d.cli.`**`initial_model`**(*opts*)

`refl1d.cli.`**`load_problem`**(*args*)

`refl1d.cli.`**`main`**()

`refl1d.cli.`**`make_store`**(*problem*, *opts*, *exists_handler*)

`refl1d.cli.`**`mesh`**(*problem*, *vars=None*, *n=40*)

`refl1d.cli.`**`preview`**(*problem*)

`refl1d.cli.`**`recall_best`**(*problem*, *path*)

`refl1d.cli.`**`remember_best`**(*fitdriver*, *problem*, *best*)

`refl1d.cli.`**`resynth`**(*fitdriver*, *problem*, *mapper*, *opts*)

`refl1d.cli.`**`run_profile`**(*problem*, *steps*)
Model execution time profiler.

Run the program with "–profile –steps=N" to generate a function profile chart breaking down the cost of evaluating N models.

Here is the findings from one profiling session:

```
23 ms total
 6 ms rendering model
 8 ms abeles
 4 ms convolution
 1 ms setting parameters and computing nllf
```

Using the GPU for abeles/convolution will only give us 2-3x speedup.

`refl1d.cli.`**`start_remote_fit`**(*problem*, *options*, *queue*, *notify*)
Queue remote fit.

`refl1d.cli.`**`store_overwrite_query`**(*path*)

`refl1d.cli.`**`store_overwrite_query_gui`**(*path*)

4.5 refl1d.dist - Non-uniform samples

`DistributionExperiment`	Compute reflectivity from a non-uniform sample.
`Weights`	Parameterized distribution for use in DistributionExperiment.

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.

DistristributionExperiment 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 *distribution* in the context of *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 *coherent* is true, then the reflectivity of the mixture is computed from the coherent sum rather than the incoherent sum.

See `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(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()

plot(*plot_shift=None*, *profile_shift=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 density 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 available 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:

```
norm      Gaussian distribution.
halfnorm  Right half of a gaussian.
triang    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.
uniform   Flat from loc to loc+scale. Use loc=edges[0], scale=edges[-1]
          to define P as uniform over the range.
```

parameters()

4.6 refl1d.errors - Plot sample profile uncertainty

`calc_distribution`	Align the sample profiles and compute the residual difference from the measured reflectivity for a set of points.
`calc_distribution_from_state`	Align the sample profiles and compute the residual difference from the
`show_distribution`	Plot the aligned profiles and the distribution of the residuals for

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.

refl1d.errors.**calc_distribution**(*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.

refl1d.errors.**calc_distribution_from_state**(*problem*, *state*, *nshown=50*, *random=False*)

Align the sample profiles and compute the residual difference from the measured reflectivity for a set of points returned from DREAM.

refl1d.errors.**show_distribution**(*profiles*, *Q*, *residuals*)

Plot the aligned profiles and the distribution of the residuals for profiles and residuals returned from calc_distribution.

4.7 refl1d.experiment - Reflectivity fitness function

`Experiment`	Theory calculator.
`ExperimentBase`	
`MixedExperiment`	Support composite sample reflectivity measurements.
`nice`	Fix v to a value with a given number of digits of precision
`plot_sample`	Quick plot of a reflectivity sample and the corresponding reflectivity.

Experiment definition

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

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

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 steps in Angstroms *dA* discretization condition 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_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_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.

`amplitude`(*resolution=False*)

Calculate reflectivity amplitude at the probe points.

`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()

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

plot_profile(*plot_shift=None*)

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

reflectivity(*resolution=True*)

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=0.10000000000000001*)

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(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``()`

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

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

`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

class refl1d.experiment.**MixedExperiment**(*samples=None*, *ratio=None*, *probe=None*, *name=None*, *coherent=False*, ***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 you 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(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()

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

plot_profile(*plot_shift=None*)

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

reflectivity(*resolution=True*)

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.

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.

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.

4.8 refl1d.fitproblem - Interface between models and fitters

`FitProblem`	
`Fitness`	
`MultiFitProblem`	Weighted fits for multiple models.
`Result`	
`TWalk`	
`fit`	Perform a fit
`load_problem`	Load a problem definition from a python script file.
`mesh`	
`preview`	Preview the models in preparation for fitting
`show_chisq`	Show chisq statistics on a drawing from the likelihood function.
`show_correlations`	List correlations between parameters in descending order.
`show_stats`	Print a stylized list of parameter names and values with range bars.

Interface between the models and the fitters.

`Fitness` defines the interface that new model definitions must follow.

`FitProblem` wraps a fitness function for use in the fitters.

`MultiFitProblem` allows simultaneous fitting of multiple functions.

WARNING within models self.parameters() returns a tree of all possible parameters associated with the model. Within fit problems, self.parameters is a list of fitted parameters only.

class `refl1d.fitproblem.`**FitProblem**(*fitness*)
Bases: `object`

chisq()
Return sum squared residuals normalized by the degrees of freedom.

In the context of a composite fit, the reduced chisq on the individual models only considers the points and the fitted parameters within the individual model.

Note that this does not include cost factors due to constraints on the parameters, such as sample_offset ~ N(0,0.01).

cov(*pvec=None*, *step=None*, *tol=1e-08*)
Return the covariance matrix inv(J'J) at point p.

We provide some protection against singular matrices by setting singular values smaller than tolerance *tol* to the tolerance value.

getp()
Returns the current value of the parameter vector.

jacobian(*pvec=None*, *step=None*)
Returns the derivative wrt the fit parameters at point p.

Numeric derivatives are calculated based on step, where step is the portion of the total range for parameter j, or the portion of point value p_j if the range on parameter j is infinite.

model_nllf()
Negative log likelihood of seeing data given model.

model_parameters()
Parameters associated with the model.

model_points()
Number of data points associated with the model.

model_reset()
Prepare for the fit.

This sets the parameters and the bounds properties that the solver is expecting from the fittable object. We also compute the degrees of freedom so that we can return a normalized fit likelihood.

If the set of fit parameters changes, then model_reset must be called.

model_update()
Update the model according to the changed parameters.

nllf(*pvec=None*)
Compute the cost function for a new parameter set p.

Note that this is not simply the sum-squared residuals, but instead is the negative log likelihood of seeing the data given the model plus the negative log likelihood of seeing the model. The individual likelihoods are scaled by 1/max(P) so that normalization constants can be ignored.

parameter_nllf()
Returns negative log likelihood of seeing parameters p.

parameter_residuals()
Returns negative log likelihood of seeing parameters p.

plot(*p=None*, *fignum=None*, *figfile=None*)

randomize()
Generates a random model.

residuals()
Return the model residuals.

restore_data()
Restore original data after resynthesis.

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

save (*basename*)

setp (*pvec*)
Set a new value for the parameters into the model. If the model is valid, calls model_update to signal that the model should be recalculated.

Returns True if the value is valid and the parameters were set, otherwise returns False.

show ()

simulate_data (*noise=None*)
Simulate data with added noise

stderr (*pvec=None*, *step=None*)
Return parameter uncertainty.

This is just the sqrt diagonal of covariance matrix inv(J'J) at point p.

valid (*pvec*)

class `refl1d.fitproblem.`**Fitness**
Bases: `object`

nllf ()
Return the negative log likelihood value of the current parameter set.

numpoints ()
Return the number of data points.

parameters ()
Return the set of parameters in the model.

plot ()
Plot the model to the current figure. You only get one figure, but you can make it as complex as you want. This will be saved as a png on the server, and composed onto a results webpage.

residiuals ()
Return residuals for current theory minus data. For levenburg-marquardt.

restore_data ()
Restore the original data in the model (after resynth).

resynth_data ()
Generate fake data based on uncertainties in the real data. For Monte Carlo resynth-refit uncertainty analysis. Bootstrapping?

save (*basename*)
Save the model to a file based on basename+extension. This will point to a path to a directory on a remote machine; don't make any assumptions about information stored on the server. Return the set of files saved so that the monitor software can make a pretty web page.

update ()
Called when parameters have been updated. Any cached values will need to be cleared and the model reevaluated.

class `refl1d.fitproblem.`**MultiFitProblem** (*models*, *weights=None*)
Bases: `refl1d.fitproblem.FitProblem`

Weighted fits for multiple models.

chisq ()
Return sum squared residuals normalized by the degrees of freedom.

In the context of a composite fit, the reduced chisq on the individual models only considers the points and the fitted parameters within the individual model.

Note that this does not include cost factors due to constraints on the parameters, such as sample_offset ~ N(0,0.01).

cov (*pvec=None*, *step=None*, *tol=1e-08*)
Return the covariance matrix inv(J'J) at point p.

We provide some protection against singular matrices by setting singular values smaller than tolerance *tol* to the tolerance value.

getp ()
Returns the current value of the parameter vector.

jacobian (*pvec=None*, *step=None*)
Returns the derivative wrt the fit parameters at point p.

Numeric derivatives are calculated based on step, where step is the portion of the total range for parameter j, or the portion of point value p_j if the range on parameter j is infinite.

model_nllf ()
Return cost function for all data sets

model_parameters ()
Return parameters from all models

model_points ()
Return number of points in all models

model_reset ()
Prepare for the fit.

This sets the parameters and the bounds properties that the solver is expecting from the fittable object. We also compute the degrees of freedom so that we can return a normalized fit likelihood.

If the set of fit parameters changes, then model_reset must be called.

model_update ()
Let all models know they need to be recalculated

nllf (*pvec=None*)
Compute the cost function for a new parameter set p.

Note that this is not simply the sum-squared residuals, but instead is the negative log likelihood of seeing the data given the model plus the negative log likelihood of seeing the model. The individual likelihoods are scaled by 1/max(P) so that normalization constants can be ignored.

parameter_nllf ()
Returns negative log likelihood of seeing parameters p.

parameter_residuals ()
Returns negative log likelihood of seeing parameters p.

plot (*fignum=1*, *figfile=None*)

randomize ()
Generates a random model.

residuals ()

restore_data ()
Restore original data after resynthesis.

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

save(*basename*)

setp(*pvec*)
Set a new value for the parameters into the model. If the model is valid, calls model_update to signal that the model should be recalculated.

Returns True if the value is valid and the parameters were set, otherwise returns False.

show()

simulate_data(*noise=None*)
Simulate data with added noise

stderr(*pvec=None*, *step=None*)
Return parameter uncertainty.

This is just the sqrt diagonal of covariance matrix inv(J'J) at point p.

valid(*pvec*)

class refl1d.fitproblem.**Result**(*problem*, *solution*)

mcmc(*samples=100000.0*, *burnin=None*, *walker=<class refl1d.fitproblem.TWalk at 0x3ce6a70>*)
Markov Chain Monte Carlo resampler.

plot()

resample(*samples=100*, *restart=False*, *fitter=None*, ***kw*)
Refit the result multiple times with resynthesized data, building up an array in Result.samples which contains the best fit to the resynthesized data. *samples* is the number of samples to generate. *fitter* is the (local) optimizer to use. The kw are the parameters for the optimizer.

save(*basename*)
Save the parameter table and the fitted model.

show()
Show the model parameters and plots

show_stats()

showmodel()

showpars()

class refl1d.fitproblem.**TWalk**(*problem*)

run(*N*, *x0*, *x1*)

refl1d.fitproblem.**fit**(*models=[]*, *weights=None*, *fitter=None*, ***kw*)
Perform a fit

refl1d.fitproblem.**load_problem**(*file*, *options=[]*)
Load a problem definition from a python script file.

sys.argv is set to `[file] + options` within the context of the script.

The user must define `problem=FitProblem(...)` within the script.

Raises ValueError if the script does not define problem.

refl1d.fitproblem.**mesh**(*models=[]*, *weights=None*, *vars=None*, *n=40*)

`refl1d.fitproblem.`**`preview`**`(`*models=[ ], weights=None*`)`
: Preview the models in preparation for fitting

`refl1d.fitproblem.`**`show_chisq`**`(`*chisq, fid=None*`)`
: Show chisq statistics on a drawing from the likelihood function.

  dof is the number of degrees of freedom, required for showing the normalized chisq.

`refl1d.fitproblem.`**`show_correlations`**`(`*pars, points, fid=None*`)`
: List correlations between parameters in descending order.

`refl1d.fitproblem.`**`show_stats`**`(`*pars, points, fid=None*`)`
: Print a stylized list of parameter names and values with range bars.

  Report mean +/- std of the samples as the parameter values.

4.9 refl1d.fitservice - Remote job plugin for fit jobs

`ServiceMonitor`	Display fit progress on the console
`fitservice`	

Fit job definition for the distributed job queue.

class `refl1d.fitservice.`**`ServiceMonitor`**`(`*problem, path, progress=60, improvement=60*`)`
: Bases: `refl1d.mystic.monitor.TimedUpdate`

  Display fit progress on the console

  `config_history``(`*history*`)`

  `show_improvement``(`*history*`)`

  `show_progress``(`*history*`)`

`refl1d.fitservice.`**`fitservice`**`(`*request*`)`

4.10 refl1d.fitters - Wrappers for various optimization algorithms

`AmoebaFit`	
`BFGSFit`	
`ConsoleMonitor`	Display fit progress on the console
`DEFit`	
`DreamFit`	
`DreamModel`	DREAM wrapper for refl1d models.
`FitBase`	
`FitDriver`	
`FitOptions`	
`MonitorRunner`	Adaptor which allows non-mystic solvers to accept progress monitors.
`MultiStart`	
`PSFit`	
`PTFit`	
`RLFit`	
`Resampler`	
`SnobFit`	
`StepMonitor`	Collect information at every step of the fit and save it to a file.

class `refl1d.fitters.`**`AmoebaFit`**(*problem*)
Bases: `refl1d.fitters.FitBase`

`solve`(*monitors=None*, *mapper=None*, ***options*)

class `refl1d.fitters.`**`BFGSFit`**(*problem*)
Bases: `refl1d.fitters.FitBase`

`solve`(*monitors=None*, *mapper=None*, ***options*)

class `refl1d.fitters.`**`ConsoleMonitor`**(*problem*, *progress=1*, *improvement=30*)
Bases: `refl1d.mystic.monitor.TimedUpdate`

Display fit progress on the console

`config_history`(*history*)

`show_improvement`(*history*)

`show_progress`(*history*)

class `refl1d.fitters.`**`DEFit`**(*problem*)
Bases: `refl1d.fitters.FitBase`

`solve`(*monitors=None*, *mapper=None*, ***options*)

class `refl1d.fitters.`**`DreamFit`**(*problem*)
Bases: `refl1d.fitters.FitBase`

`error_plot`(*figfile*)

`plot`(*output_path*)

`save`(*output_path*)

`show`()

`solve`(*monitors=None*, *mapper=None*, ***options*)

class `refl1d.fitters.`**`DreamModel`**(*problem=None*, *mapper=None*)
Bases: `dream.model.MCMCModel`

DREAM wrapper for refl1d models.

`log_density`(*x*)

`map`(*pop*)

`nllf`(*x*)
Negative log likelihood of seeing models given parameters *x*

`plot`(*x*)

class `refl1d.fitters.`**`FitBase`**(*problem*)
Bases: `object`

`solve`(*monitors=None*, *mapper=None*, ***options*)

class `refl1d.fitters.`**`FitDriver`**(*fitclass=None*, *problem=None*, *monitors=None*, *mapper=None*, ***options*)
Bases: `object`

`fit`()

`plot`(*output_path*)

`save`(*output_path*)

`show`()

class `refl1d.fitters.`**`FitOptions`**(*fitclass*)

Bases: `object`

`set_from_cli`(*opts*)

class `refl1d.fitters.`**`MonitorRunner`**(*monitors*, *problem*)

Bases: `object`

Adaptor which allows non-mystic solvers to accept progress monitors.

class `refl1d.fitters.`**`MultiStart`**(*fitter*)

Bases: `refl1d.fitters.FitBase`

`solve`(*monitors=None*, *mapper=None*, ***options*)

class `refl1d.fitters.`**`PSFit`**(*problem*)

Bases: `refl1d.fitters.FitBase`

`solve`(*monitors=None*, *mapper=None*, ***options*)

class `refl1d.fitters.`**`PTFit`**(*problem*)

Bases: `refl1d.fitters.FitBase`

`solve`(*monitors=None*, *mapper=None*, ***options*)

class `refl1d.fitters.`**`RLFit`**(*problem*)

Bases: `refl1d.fitters.FitBase`

`solve`(*monitors=None*, *mapper=None*, ***options*)

class `refl1d.fitters.`**`Resampler`**(*fitter*)

Bases: `refl1d.fitters.FitBase`

`solve`(***options*)

class `refl1d.fitters.`**`SnobFit`**(*problem*)

Bases: `refl1d.fitters.FitBase`

`solve`(*monitors=None*, *mapper=None*, ***options*)

class `refl1d.fitters.`**`StepMonitor`**(*problem, fid, fields=['step', 'time', 'value', 'point']*)

Bases: `refl1d.mystic.monitor.Monitor`

Collect information at every step of the fit and save it to a file.

fid is the file to save the information to *fields* is the list of "step|time|value|point" fields to save

The point field should be last in the list.

`config_history`(*history*)

4.11 refl1d.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 `refl1d.freeform.`**`FreeInterface`**(*interface=0*, *below=None*, *above=None*, *dz=None*, *dp=None*, *name='Interface'*)

Bases: `refl1d.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

parameters()

render(*probe*, *slabs*)

thickness

class refl1d.freeform.**FreeLayer**(*thickness=0*, *left=None*, *right=None*, *rho=[ ]*, *irho=[ ]*, *rhoz=[ ]*, *irhoz=[ ]*, *name='Freeform'*)
Bases: refl1d.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.

constraints()
Constraints

find(*z*)
Find the layer at depth z.

Returns layer, start, end

parameters()

render(*probe*, *slabs*)

class refl1d.freeform.**FreeformInterface01**(*thickness=0*, *interface=0*, *below=None*, *above=None*, *z=None*, *vf=None*, *name='Interface'*)
Bases: refl1d.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.

constraints()
Constraints

find(*z*)
Find the layer at depth z.

Returns layer, start, end

parameters()

render(*probe*, *slabs*)

4.12 refl1d.fresnel - Pure python Fresnel reflectivity calculator

`Fresnel`	Function for computing the Fresnel reflectivity for a single interface.
`test`	

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`**`()`

4.13 refl1d.garefl - Adaptor for garefl models

`load`

Load garefl models into refl1d

`refl1d.garefl.`**`load`**`(`*modelfile*`)`

4.14 refl1d.initpop - Population initialization strategies

`lhs_init`	Latin Hypercube Sampling
`cov_init`	Initialize *N* sets of random variables from a gaussian model.
`random_init`	Generate a random population from the problem parameters.

Population initialization routines.

To start the analysis an initial population is required. This will be an array of size M x N, where M is the number of dimensions in the fitting problem and N is the number of individuals in the population.

Three functions are provided:

1. lhs_init(N, pars) returns a latin hypercube sampling, which tests every parameter at each of N levels.

2. cov_init(N, pars, cov) returns a Gaussian sample along the ellipse defined by the covariance matrix, cov. Covariance defaults to diag(dx) if dx is provided as a parameter, or to I if it is not.

3. rand_init(N, pars) returns a random population following the prior distribution of the parameter values.

Additional options are random box: rand(M,N) or random scatter: randn(M,N).

`refl1d.initpop.`**`lhs_init`**(*N*, *pars*, *include_current=False*)

Latin Hypercube Sampling

Returns an array whose columns and rows each have *N* samples from equally spaced bins between *bounds*=(xmin, xmax) for the column. Unlike random, this method guarantees a certain amount of coverage of the parameter space. Consider, though that the diagonal matrix satisfies the LHS condition, and you can see that the guarantees are not very strong. A better methods, similar to sudoku puzzles, would guarantee coverage in each block of the matrix, but this is not yet implmeneted.

If include_current is True, then the current value of the parameters is returned as the first point in the population, preserving the the LHS property.

Note: Indefinite ranges are not supported.

`refl1d.initpop.`**`cov_init`**(*N*, *pars*, *include_current=False*, *cov=None*, *dx=None*)

Initialize *N* sets of random variables from a gaussian model.

The center is at *x* with an uncertainty ellipse specified by the 1-sigma independent uncertainty values *dx* or the full covariance matrix uncertainty *cov*.

For example, create an initial population for 20 sequences for a model with local minimum x with covariance matrix C:

```
pop = cov_init(cov=C, pars=p, N=20)
```

If include_current is True, then the current value of the parameters is returned as the first point in the population.

`refl1d.initpop.`**`random_init`**(*N*, *pars*, *include_current=False*)

Generate a random population from the problem parameters.

Values are selected at random from the bounds of the problem using a uniform distribution. A certain amount of clustering is expected using this method.

If include_current is True, then the current value of the parameters is returned as the first point in the population.

4.15 refl1d.instrument - Reflectivity instrument definition

`Monochromatic`	Instrument representation for scanning reflectometers.
`Pulsed`	Instrument representation for pulsed reflectometers.

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

4.15.1 Scanning Reflectometers

`refl1d.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 $\Delta\theta/\theta$ while opening.

```
>>> 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:

```
>>> 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:

```
>>> probe2 = geometry.probe(Q=Q, data=(R,dR), sample_broadening=0.1)
```

For magnetic systems a polarized beam probe is needed:

```
>>> 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:

```
>>> print geometry
== Instrument SP:2 ==
radiation = neutron at 5.0042 Angstrom with 0.9% resolution
slit distances = 2086 mm and 230 mm
fixed region below 0.5 and above 90 degrees
slit openings at Tlo are 0.2 mm
sample width = 1e+10 mm
sample broadening = 0 degrees
```

4.15.2 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:

```
>>> 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:

```
>>> 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:

```
>>> print SP2.defaults()
== Instrument class SP:2 ==
radiation = neutron at 5.0042 Angstrom with 0.9% resolution
slit distances = 2086 mm and 230 mm
```

4.15.3 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.

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

`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.

classmethod `defaults`()

Return default instrument properties as a printable string.

`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 *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.

`resolution`(***kw*)

Calculate resolution at each angle.

Return

T*, *dT [[float] | °] Angles and angular divergence.

L*, *dL [[float] | Å] Wavelengths and wavelength dispersion.

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)] usabe range of times for TOF data

T [float | °] 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 | °] 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 | ° 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

`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 *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.

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

`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.

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

`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.

4.16 refl1d.magnetic - Magnetic Models

`FreeMagnetic`	Linear change in magnetism throughout layer.
`MagneticLayer`	
`MagneticSlab`	Region of constant magnetism.
`MagneticTwist`	Linear change in magnetism throughout layer.

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 is not This could in principle be approximated from the dipole moments of the individual moments aligned within the sample, then you would see the If the fields for all carriers are aligned with the polarization direction, you will see the idealized magnetic scattering strength will see the saturated This is determined by the number and strength of the magnetic 'carriers', the amount of order, and the direct or :math:' ho_M cos(heta_M)', where orientation, which leads to over-parameterization in the fits. The reflectometry technique is sensitive

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

class `refl1d.magnetic.`**`FreeMagnetic`**(*stack*, *z=[ ]*, *rhoM=[ ]*, *thetaM=[ ]*, *name='freemag'*, ***kw*)

Bases: `refl1d.magnetic.MagneticLayer`

Linear change in magnetism throughout layer.

`constraints`()

Constraints

`find`(*z*)

Find the layer at depth z.

Returns layer, start, end

`parameters`()

`profile`(*Pz*)

`render`(*probe*, *slabs*)

`render_stack`(*probe*, *slabs*)

Render the nuclear sld structure.

If either the interface below or the interface above is left unspecified, the corresponding nuclear interface is used.

Returns the anchor point in the nuclear structure and interface widths at either end of the magnetic slab.

`thickness`

Thickness of the magnetic region

`thicknessM`

class `refl1d.magnetic.`**`MagneticLayer`**(*stack=None*, *dead_below=0*, *dead_above=0*, *interface_below=None*, *interface_above=None*, *name='magnetic'*)

Bases: `refl1d.model.Layer`

`constraints`()

Constraints

`find`(*z*)

Find the layer at depth z.

Returns layer, start, end

`parameters`()

`render`(*probe*, *slabs*)

Use the probe to render the layer into a microslab representation.

`render_stack`(*probe*, *slabs*)

Render the nuclear sld structure.

If either the interface below or the interface above is left unspecified, the corresponding nuclear interface is used.

Returns the anchor point in the nuclear structure and interface widths at either end of the magnetic slab.

`thickness`

Thickness of the magnetic region

`thicknessM`

class `refl1d.magnetic.`**`MagneticSlab`**(*stack*, *rhoM=0*, *thetaM=270*, *name='magnetic'*, ***kw*)

Bases: `refl1d.magnetic.MagneticLayer`

Region of constant magnetism.

`constraints`()

Constraints

`find`(*z*)

Find the layer at depth z.

Returns layer, start, end

`parameters`()

`render`(*probe*, *slabs*)

`render_stack`(*probe*, *slabs*)

Render the nuclear sld structure.

If either the interface below or the interface above is left unspecified, the corresponding nuclear interface is used.

Returns the anchor point in the nuclear structure and interface widths at either end of the magnetic slab.

`thickness`

Thickness of the magnetic region

`thicknessM`

class `refl1d.magnetic.`**`MagneticTwist`**(*stack*, *rhoM=[0, 0]*, *thetaM=[270, 270]*, *name='twist'*, ***kw*)

Bases: `refl1d.magnetic.MagneticLayer`

Linear change in magnetism throughout layer.

constraints()
Constraints

find(*z*)
Find the layer at depth z.

Returns layer, start, end

parameters()

render(*probe*, *slabs*)

render_stack(*probe*, *slabs*)
Render the nuclear sld structure.

If either the interface below or the interface above is left unspecified, the corresponding nuclear interface is used.

Returns the anchor point in the nuclear structure and interface widths at either end of the magnetic slab.

thickness
Thickness of the magnetic region

thicknessM

4.17 refl1d.mapper - Parallel processing implementations

AMQPMapper	
MPMapper	
SerialMapper	
nice	
setpriority	Set The Priority of a Windows Process. Priority is a value between 0-5

class refl1d.mapper.**AMQPMapper**
Bases: object

static start_mapper(*problem*, *modelargs*)

static start_worker(*problem*)

static stop_mapper(*mapper*)

class refl1d.mapper.**MPMapper**
Bases: object

static start_mapper(*problem*, *modelargs*, *cpus=None*)

static start_worker(*problem*)

static stop_mapper(*mapper*)

class refl1d.mapper.**SerialMapper**
Bases: object

static start_mapper(*problem*, *modelargs*)

static start_worker(*problem*)

static stop_mapper(*mapper*)

refl1d.mapper.**nice**()

`refl1d.mapper.`**`setpriority`**(*pid=None*, *priority=1*)

Set The Priority of a Windows Process. Priority is a value between 0-5 where 2 is normal priority and 5 is maximum. Default sets the priority of the current python process but can take any valid process ID.

4.18 refl1d.material - Material

`Material`	Description of a solid block of material.
`Mixture`	Mixed block of material.
`SLD`	Unknown composition.
`Vacuum`	Empty layer
`Scatterer`	A generic scatterer separates the lookup of the scattering factors from the calculation of the scattering length density.
`ProbeCache`	Probe proxy for materials properties.

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.

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.

density : float | g·cm^{-3}

If specified, set the bulk density for the material.

natural_density : float | g·cm^{-3}

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

use_incoherent = False : boolean

True if incoherent scattering should be interpreted as absorption.

fitby = 'bulk_density' : string

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

value : Parameter or float | 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:

```
>>> m = Material('Pd', fitby='cell_volume')
>>> m.cell_volume.range(1,10)
Parameter(Pd cell volume)
>>> print m.density.value, m.cell_volume.value
12.02 14.7017085944
```

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

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 `WrappedParameter` for *density* is another alternative.

parameters()

sld(*probe*)

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 $Nr_e(f_1 + jf_2)$ 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 uses 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

`parameters`()

`sld`(*probe*)

class `refl1d.material.`**`Scatterer`**

Bases: `object`, `refl1d.model._MaterialStacker`

A generic scatterer separates the lookup of the scattering factors from the calculation of the scattering length density. This allows programs to fit density and alloy composition more efficiently.

Note: the Scatterer base class is extended by `_MaterialStacker` so that materials can be implicitly converted to slabs when used in stack construction expressions. It is not done directly to avoid circular dependencies between `model` and `material`.

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

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.

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 cash.

`clear`()

`scattering_factors`(*material*)
Return the scattering factors for the material, retrieving them from the cache if they have already been looked up.

4.19 refl1d.materialdb - Materials Database

`air`	Empty layer
`water`	Description of a solid block of material.
`H2O`	Description of a solid block of material.
`heavywater`	Description of a solid block of material.
`D2O`	Description of a solid block of material.
`lightheavywater`	Description of a solid block of material.
`DHO`	Description of a solid block of material.
`silicon`	Description of a solid block of material.
`Si`	Description of a solid block of material.
`sapphire`	Description of a solid block of material.
`Al2O3`	Description of a solid block of material.
`gold`	Description of a solid block of material.
`Au`	Description of a solid block of material.
`permalloy`	Description of a solid block of material.
`Ni8Fe2`	Description of a solid block of material.

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:

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

4.20 refl1d.model - Reflectivity Models

`Repeat`	Repeat a layer or stack.
`Slab`	A block of material.
`Stack`	Reflectometry layer stack
`Layer`	Component of a material description.

Reflectometry models

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

class `refl1d.model.`**`Repeat`**(*stack*, *repeat=1*, *interface=None*, *name=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.

`constraints`()

Constraints

`find`(*z*)

Find the layer at depth z.

Returns layer, start, end

`magnetic`

`parameters`()

`render`(*probe*, *slabs*)

`thickness`

class `refl1d.model.`**`Slab`**(*material=None*, *thickness=0*, *interface=0*, *name=None*)

Bases: `refl1d.model.Layer`

A block of material.

`constraints`()

Constraints

`find`(*z*)

Find the layer at depth z.

Returns layer, start, end

`parameters`()

`render`(*probe*, *slabs*)

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 overlaying 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 accommodate. You cannot make nested stacks.

magnetic

parameters()

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 (Interface function) Interface for the top of the layer.

constraints()
Constraints

find(*z*)
Find the layer at depth z.

Returns layer, start, end

parameters()
Returns a list of parameters used in the layer.

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

4.21 refl1d.mono - Freeform - Monotonic Spline

FreeInterface	A freeform section of the sample modeled with monotonic splines.
FreeLayer	A freeform section of the sample modeled with splines.
count_inflections	Count the number of inflection points in the spline curve
hermite	Computes the cubic hermite polynomial p(xt).
inflections	
monospline	Monotonic cubic hermite interpolation.
plot_inflection	

Monotonic spline modeling for free interfaces

class refl1d.mono.**FreeInterface**(*thickness=0*, *interface=0*, *below=None*, *above=None*, *dz=None*, *dp=None*, *name='Interface'*)
Bases: refl1d.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

parameters()

profile(*Pz*)

render(*probe*, *slabs*)

class refl1d.mono.**FreeLayer**(*below=None*, *above=None*, *thickness=0*, *z=[]*, *rho=[]*, *irho=[]*, *name='Freeform'*)
Bases: refl1d.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

parameters()

profile(*Pz*, *below*, *above*)

render(*probe*, *slabs*)

refl1d.mono.**count_inflections**(**args*, ***kw*)
Count the number of inflection points in the spline curve

refl1d.mono.**hermite**(**args*, ***kw*)
Computes the cubic hermite polynomial p(xt).

The polynomial goes through all points (x_i,y_i) with slope m_i at the point.

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

refl1d.mono.**monospline**(**args*, ***kw*)
Monotonic cubic hermite interpolation.

Returns $p(x_t)$ where $p(x_i) = y_i$ and $p(x) \leq p(xi)$ if $y_i \leq y_{i+1}$ for all y_i. Also works for decreasing values y, resulting in decreasing $p(x)$. If y is not monotonic, then $p(x)$ may peak higher than any y, so this function is not suitable for a strict constraint on the interpolated function when y values are unconstrained.

http://en.wikipedia.org/wiki/Monotone_cubic_interpolation

refl1d.mono.**plot_inflection**(*x*, *y*)

4.22 refl1d.ncnrdata - NCNR Data

ANDR	Instrument definition for NCNR AND/R diffractometer/reflectometer.
NCNRData	
NG1	Instrument definition for NCNR NG-1 reflectometer.
NG7	Instrument definition for NCNR NG-7 reflectometer.
XRay	Instrument definition for NCNR X-ray reflectometer.
find_xsec	Find files containing the polarization cross-sections.
load	Return a probe for NCNR data.
load_magnetic	Return a probe for magnetic NCNR data.
parse_file	Parse NCNR reduced data file returning *header* and *data*.

NCNR data loaders

The following instruments are defined:

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:

```
>>> 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:

```
>>> 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:

```
>>> instrument = NCNR.ANDR(Tlo=0.5, slits_at_Tlo=0.2, slits_below=0.1)
>>> probe = instrument.probe(T=numpy.linspace(0,5,51))
>>> probe.plot_resolution()
>>> sample = silicon(0,10) | gold(100,10) | air
>>> M = Experiment(probe=probe, sample=sample)
>>> M.simulate_data() # Optional
>>> M.plot()
```

And for magnetic:

```
>>> instrument = NCNR.NG1(slits_at_Tlo=1)
>>> #sample = silicon(0,10) | Magnetic(permalloy(100,10),rho_M=3) | air
>>> #M = Experiment(probe=probe, sample=sample)
>>> #M.simulate_data()
>>> #M.plot()
>>> #probe = instrument.simulate_magnetic(sample, T=numpy.linspace(0,5,51))
>>> #h = pylab.plot(probe.Q, probe.dQ)
>>> #h = pylab.ylabel('resolution (1-sigma)')
>>> #h = pylab.xlabel('Q (inv A)')
```

See `instrument` for details.

class `refl1d.ncnrdata.`**`ANDR`**`(`***kw*`)`

Bases: `refl1d.ncnrdata.NCNRData`, `refl1d.instrument.Monochromatic`

Instrument definition for NCNR AND/R diffractometer/reflectometer.

`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.

classmethod **`defaults`**`()`

Return default instrument properties as a printable string.

`load``(`*filename*, ***kw*`)`

`load_magnetic``(`*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 *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.

`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.

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.

`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.

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

`load`(*filename*, ***kw*)

`load_magnetic`(*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 *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.

`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.

class `refl1d.ncnrdata.`**`NG7`**(***kw*)
Bases: `refl1d.ncnrdata.NCNRData`, `refl1d.instrument.Monochromatic`

Instrument definition for NCNR NG-7 reflectometer.

`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.

classmethod `defaults`()

Return default instrument properties as a printable string.

`load`(*filename*, ***kw*)

`load_magnetic`(*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 *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.

`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.

class `refl1d.ncnrdata.`**`XRay`**(***kw*)

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 sample_broadening to define the entire divergence:

```
>>> from refl1d.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
```

`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.

classmethod `defaults`()

Return default instrument properties as a printable string.

`load`(*filename*, ***kw*)

load_magnetic (*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 *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.

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.

refl1d.ncnrdata.**find_xsec** (*filename*)

Find files containing the polarization cross-sections.

Returns tuple with file names for ++ +- -+ – cross sections, or None if the spin cross section does not exist.

refl1d.ncnrdata.**load** (*filename*, *instrument=None*, ***kw*)

Return a probe for NCNR data.

Keyword arguments are as specified Monochromatic instruments.

refl1d.ncnrdata.**load_magnetic** (*filename*, *Aguide=270*, *shared_beam=True*, ***kw*)

Return a probe for magnetic NCNR data.

***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 the be the file name for the cross section or None if the cross section does not exist.

***Aguide* (degrees)** Angle of the guide field relative to the beam. 270 is the default.

***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 `instrument.Monochromatic`.

The data sets should are 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_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.

4.23 refl1d.numpyerrors - Decorator for function level error behaviour

`errors`
`ignored`
`printed`
`raised`
`warned`

Decorator for handling numpy errors.

Use this when you have a routine with numeric issues such as divide by zero which are known to be harmless, for example, because infinite or NaN results are allowed by the interface, or because the remainder of the code accommodates the exceptional conditions.

4.23.1 Usage

This is a wrapper around the numpy.seterr() command, and uses the same types of error handling controls, but in a with context or as a decorator:

```
with Errors(...):
    statements

@errors(...)
def f():
    statements
```

The arguments to Errors and errors are identical to numpy.seterr.

Some convenience decorators are predefined: ignored, raised, printed, warned.

4.23.2 Example

```
>>> import numpy
>>> with numpy.errstate(all='ignore'): x = 1/numpy.zeros(3)
>>> with numpy.errstate(all='print'): x = 1/numpy.zeros(3)
Warning: divide by zero encountered in divide
>>> @ignored
... def f(): x = 1/numpy.zeros(3)
>>> f()
>>> @printed
... def g(): x = 1/numpy.zeros(3)
>>> g()
Warning: divide by zero encountered in divide
```

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

`refl1d.numpyerrors.`**`ignored`**`(`*f*`)`

`refl1d.numpyerrors.`**`printed`**`(`*f*`)`

`refl1d.numpyerrors.`**`raised`**`(`*f*`)`

`refl1d.numpyerrors.`**`warned`**`(`*f*`)`

4.24 refl1d.partemp - Parallel tempering optimizer

`parallel_tempering`	Perform a MCMC walk using multiple temperatures in parallel.

Parallel tempering for continuous function optimization and uncertainty analysis

The program performs Markov chain Monte Carlo exploration of a probability density function using a combination of random and differential evolution updates.

`refl1d.partemp.`**`parallel_tempering`**`(`*nllf, p, bounds, T=None, steps=1000, CR=0.90000000000000002, burn=1000, monitor=<function every_ten at 0x48d1d70>, logfile=None*`)`

Perform a MCMC walk using multiple temperatures in parallel.

Parameters

nllf [function(vector) -> float] Negative log likelihood function to be minimized. $\chi^2/2$ is a good choice for curve fitting with no prior restraints on the possible input parameters.

p [vector] Initial value

bounds [vector, vector] Box constraints on the parameter values. No support for indefinite or semi-definite programming at present

T [vector | 0 < T[0] < T[1] < ...] Temperature vector. Something like logspace(-1,1,10) will give you 10 logarithmically spaced temperatures between 0.1 and 10. The maximum temperature T[-1] determines the size of the barriers that can be easily jumped. Note that the number of temperature values limits the amount of parallelism available in the algorithm, so it may gather statistics more quickly, though it will not necessarily converge any faster.

***steps* = 1000** [int] Length of the accumulation vector. The returned history will store this many values for each temperature. These values can be used in a weighted histogram to determine parameter uncertainty.

***burn* = 1000** [int | [0,inf)] Number of iterations to perform in addition to steps. Only the last *steps* points will be preserved for each temperature. Since the value should be in the same order as *steps* to be sure that the full history is acquired.

***CR* = 0.9** [float | [0,1]] Cross-over ratio. This is the differential evolution crossover ratio to use when computing step size and direction. Use a small value to step through the dimensions one at a time, or a large value to step through all at once.

***monitor* = every_ten** [function(int step, vector x, float fx) -> None] Function to called at every iteration with the step number the best point and the best value.

***logfile* = None** [string] Name of the file which will log the history of every accepted step. Note that this includes all of the burn steps, so it can get very large.

Returns

history [History] Structure containing *best*, *best_point* and *buffer*. *best* is the best nllf value seen and *best_point* is the parameter vector which yielded *best*. The list *buffer* contains lists of tuples (step, temperature, nllf, x) for each temperature.

4.25 refl1d.polymer - Polymer models

`PolymerBrush`	Polymer brushes in a solvent
`VolumeProfile`	Generic volume profile function
`layer_thickness`	Return the thickness of a layer given the microslab z points.
`smear`	Gaussian smearing

Layer models for polymer systems.

Analytic Self-consistent Field (SCF) profile [1] [2]

layer = TetheredPolymer(polymer,solvent,head,tail,power,

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:

$$V(z) = \begin{cases} V_o & \text{if } z <= z_o \\ V_o(1-((z-z_o)/L)^2)^p & \text{if } z_o < z < z_o + L \\ 0 & \text{if } z >= z_o + L \end{cases}$$

$$V_\sigma(z) = V(z) \star \frac{e^{-\frac{1}{2}(z/\sigma)^2}}{\sqrt{2\pi\sigma^2}}$$

$$\rho(z) = \rho_p V_\sigma(z) + \rho_s(1 - V_\sigma(z))$$

[1] Zhulina, EB; Borisov, OV; Pryamitsyn, VA; Birshtein, TM (1991) "Coil-Globule Type Transitions in Polymers. 1. Collapse of Layers of Grafted Polymer Chains", Macromolecules 24, 140-149.

[2] Karima, A; Douglas, JF; Horkay, F; Fetters, LJ; Satija, SK (1996) "Comparative swelling of gels and polymer brush layers", Physica B 221, 331-336. DOI:10.1016/0921-4526(95)00946-9

where $V_\sigma(z)$ is volume fraction convoluted with brush roughness σ 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

parameters()

profile(*z*)

render(*probe*, *slabs*)

class refl1d.polymer.**VolumeProfile**(*thickness=0*, *interface=0*, *name='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 * profile + solvent.sld * (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*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 `layer_thickness()`.

constraints()
    Constraints

find(*z*)
    Find the layer at depth z.

    Returns layer, start, end

parameters()

render(*probe*, *slabs*)

refl1d.polymer.**layer_thickness**(*z*)

Return the thickness of a layer given the microslab z points.

The layer is sliced into bins of equal width, with the final bin making up the remainder. The z values given to the profile function are the centers of these bins. Using this, we can guess that the total layer thickness will be the following:

```
    2*z[-1]-z[-2] if len(z) > 0 else 2*z[0]
```

`refl1d.polymer.`**`smear`**(*z*, *P*, *sigma*)
Gaussian smearing

Parameters

***z* | vector** equally spaced sample times

***P* | vector** sample values

***sigma* | real** root-mean-squared convolution width

Returns

***Ps* | vector** smeared sample values

4.26 refl1d.probe - Instrument probe

`NeutronProbe`	
`PolarizedNeutronProbe`	Polarized neutron probe
`PolarizedNeutronQProbe`	
`Probe`	Defines the incident beam used to study the material.
`ProbeSet`	
`QProbe`	A pure Q,R probe
`Qmeasurement_union`	Determine the unique (T,dT,L,dL) across all datasets.
`XrayProbe`	X-Ray probe.
`make_probe`	Return a reflectometry measurement object of the given resolution.
`measurement_union`	Determine the unique (T,dT,L,dL) across all datasets.
`spin_asymmetry`	Compute spin asymmetry for R++, R–.

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-guide‘_** 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*)
Bases: `refl1d.probe.Probe`

`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:

dtheta [float | degrees] uncertainty in alignment angle

`apply_beam`(*calc_Q*, *calc_R*, *resolution=True*)
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 θ for a fixed wavelength $< \lambda >$, the average of all wavelengths in the probe.

Specifically:

$$\begin{aligned}
Q_c^2 &= 16\pi(\rho - \rho_{\text{incident}}) \\
Q_i &= Q_c - \delta_i Q_c(i - (n-1)/2) \qquad \text{for } i \in 0 \ldots n-1 \\
\lambda_i &=< \lambda > \\
\theta_i &= \sin^{-1}(Q_i \lambda_i / 4\pi)
\end{aligned}$$

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.

data

fresnel (*substrate=None*, *surface=None*)
Compute the 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.

name

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 reflectivity

plot_Q4 (***kwargs*)
Plot the Q**4 reflectivity associated with the probe.

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 reflectivity associated with the probe.

plot_linear (***kwargs*)
Plot the data associated with probe.

plot_log (***kwargs*)
Plot the data associated with probe.

plot_residuals (*theory=None*, *suffix=''*, *label=None*, *plot_shift=None*, ***kwargs*)

plot_resolution (*suffix=''*, *label=None*, ***kwargs*)

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 ~ 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=None*, *surface=None*)
Save the data and theory to a file.

scattering_factors (*material*)
Returns the scattering factors associated with the material given the range of wavelengths/energies used in the probe.

simulate_data (*theory*, *noise=2*)
Set the data for the probe to R, adding random noise dR.

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.

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*, *Aguide=270*)

Bases: `object`

Polarized neutron probe

xs (4 x NeutronProbe) is a sequence pp, pm, mp and mm. *Aguide* (degrees) is the angle of the guide field

`apply_beam`(*Q*, *R*, *resolution=True*)

Apply factors such as beam intensity, background, backabsorption, and footprint to the data.

`calc_Q`

`fresnel`(**args*, ***kw*)

Compute the 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`

`name`

`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 reflectivity

`plot_Q4`(***kwargs*)

`plot_SA`(*theory=None*, *label=None*, *plot_shift=None*, ***kwargs*)

`plot_fresnel`(***kwargs*)

`plot_linear`(***kwargs*)

`plot_log`(***kwargs*)

`plot_residuals`(***kwargs*)

`plot_resolution`(***kwargs*)

pm

pp

restore_data()
Restore the original data.

resynth_data()
Generate new data according to the model R ~ 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=None*, *surface=None*)
Save the data and theory to a file.

scattering_factors(*material*)
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.

class refl1d.probe.**PolarizedNeutronQProbe**(*xs=None*)
Bases: refl1d.probe.PolarizedNeutronProbe

apply_beam(*Q*, *R*, *resolution=True*)
Apply factors such as beam intensity, background, backabsorption, and footprint to the data.

calc_Q

fresnel(**args*, ***kw*)
Compute the 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

name

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 reflectivity

plot_Q4(***kwargs*)

plot_SA(*theory=None*, *label=None*, *plot_shift=None*, ***kwargs*)

plot_fresnel(***kwargs*)

plot_linear(***kwargs*)

plot_log(***kwargs*)

plot_residuals(***kwargs*)

plot_resolution(***kwargs*)

pm

pp

restore_data()
Restore the original data.

resynth_data()
Generate new data according to the model R ~ 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=None*, *surface=None*)
Save the data and theory to a file.

scattering_factors(*material*)
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.

class `refl1d.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*)

Bases: `object`

Defines the incident beam used to study the material.

For calculation purposes, probe needs to return the values Q_{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.

Measurement properties:

> *intensity* is the beam intensity *background* is the background *back_absorption* is the amount of absorption through the substrate *theta_offset* is the offset of the sample from perfect alignment *back_reflectivity* is 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 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:

> *substrate* is the material which makes up the substrate *surface* is the material which makes up the surface *view* is 'fresnel', 'log', 'linear', 'q4', 'residual' *plot_shift* is the number of pt to shift each new dataset

Normally *view* is set directly in the class rather than the instance since it is not specific to the view. The fresnel substrate and surface materials are a property of the sample, and should share the same material.

`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:

dtheta [float | degrees] uncertainty in alignment angle

`apply_beam`(*calc_Q*, *calc_R*, *resolution=True*)

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 θ for a fixed wavelength $< \lambda >$, the average of all wavelengths in the probe.

Specifically:

$$\begin{aligned} Q_c^2 &= 16\pi(\rho - \rho_{\text{incident}}) \\ Q_i &= Q_c - \delta_i Q_c (i - (n-1)/2) \qquad \text{for } i \in 0 \ldots n-1 \\ \lambda_i &= < \lambda > \\ \theta_i &= \sin^{-1}(Q_i \lambda_i / 4\pi) \end{aligned}$$

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.

data

fresnel (*substrate=None*, *surface=None*)
Compute the 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.

name

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 reflectivity

plot_Q4 (***kwargs*)
Plot the Q**4 reflectivity associated with the probe.

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 reflectivity associated with the probe.

plot_linear(***kwargs*)
Plot the data associated with probe.

plot_log(***kwargs*)
Plot the data associated with probe.

plot_residuals(*theory=None*, *suffix=''*, *label=None*, *plot_shift=None*, ***kwargs*)

plot_resolution(*suffix=''*, *label=None*, ***kwargs*)

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 ~ 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=None*, *surface=None*)
Save the data and theory to a file.

scattering_factors(*material*)
Returns the scattering factors associated with the material given the range of wavelengths/energies used in the probe.

simulate_data(*theory*, *noise=2*)
Set the data for the probe to R, adding random noise dR.

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.

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*)
Bases: refl1d.probe.Probe

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:

dtheta [float | degrees] uncertainty in alignment angle

`apply_beam`(*calc_Q*, *calc_R*, *resolution=True*, ***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 $Q_c \pm \delta Q_c$ by varying angle θ for a fixed wavelength $< \lambda >$, the average of all wavelengths in the probe.

Specifically:

$$\begin{aligned}
Q_c^2 &= 16\pi(\rho - \rho_{\text{incident}}) \\
Q_i &= Q_c - \delta_i Q_c (i - (n-1)/2) \qquad \text{for } i \in 0 \ldots n-1 \\
\lambda_i &= < \lambda > \\
\theta_i &= \sin^{-1}(Q_i \lambda_i / 4\pi)
\end{aligned}$$

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.

`data`

`fresnel`(**args*, ***kw*)

Compute the 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.

`name`

`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()

plot(*theory=None*, ***kw*)
Plot theory against data.

Need substrate/surface for Fresnel reflectivity

plot_Q4(*theory=None*, ***kw*)
Plot the Q**4 reflectivity associated with the probe.

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 reflectivity associated with the probe.

plot_linear(*theory=None*, ***kw*)
Plot the data associated with probe.

plot_log(*theory=None*, ***kw*)
Plot the data associated with probe.

plot_residuals(*theory=None*, ***kw*)

plot_resolution(***kw*)

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 ~ 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=None*, *surface=None*)
Save the data and theory to a file.

scattering_factors(*material*)
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.

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:

$$\hat{Q} = \sum Q_k/n$$

Then for each dataset k, find the interval $[i, i+1]$ containing the value Q, and use it to compute interpolated value for R:

$$w = (\hat{Q} - Q_i)/(Q_{i+1} - Q_i)$$
$$\hat{R} = wR_{i+1} + (1-w)R_{i+1}$$
$$\hat{\sigma}_R = \sqrt{w^2\sigma^2_{R_i} + (1-w)^2\sigma^2_{R_{i+1}}}/n$$

Average the resulting R using Gaussian error propagation:

$$\hat{R} = \sum \hat{R}_k/n$$
$$\hat{\sigma}_R = \sqrt{\sum \hat{\sigma}^2_{R_k}}/n$$

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.

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*, *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.

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:

dtheta [float | degrees] uncertainty in alignment angle

`apply_beam`(*calc_Q*, *calc_R*, *resolution=True*)
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 θ for a fixed wavelength $< \lambda >$, the average of all wavelengths in the probe.

Specifically:

$$\begin{aligned}
Q_c^2 &= 16\pi(\rho - \rho_{\text{incident}}) \\
Q_i &= Q_c - \delta_i Q_c (i - (n-1)/2) \qquad \text{for } i \in 0 \ldots n-1 \\
\lambda_i &= < \lambda > \\
\theta_i &= \sin^{-1}(Q_i \lambda_i / 4\pi)
\end{aligned}$$

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.

`data`

`fresnel`(*substrate=None*, *surface=None*)
Compute the 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.

`name`

`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 reflectivity

plot_Q4(***kwargs*)
Plot the Q**4 reflectivity associated with the probe.

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 reflectivity associated with the probe.

plot_linear(***kwargs*)
Plot the data associated with probe.

plot_log(***kwargs*)
Plot the data associated with probe.

plot_residuals(*theory=None*, *suffix=''*, *label=None*, *plot_shift=None*, ***kwargs*)

plot_resolution(*suffix=''*, *label=None*, ***kwargs*)

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 ~ 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=None*, *surface=None*)
Save the data and theory to a file.

scattering_factors(*material*)
Returns the scattering factors associated with the material given the range of wavelengths/energies used in the probe.

simulate_data(*theory*, *noise=2*)
Set the data for the probe to R, adding random noise dR.

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.

`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.`**`Qmeasurement_union`**(*xs*)
Determine the unique (T,dT,L,dL) across all datasets.

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*)
Bases: `refl1d.probe.Probe`

X-Ray probe.

Contains information about the kind of probe used to investigate the sample.

X-ray data is traditionally recorded by angle and energy, rather than angle and wavelength as is used by neutron probes.

`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:

dtheta [float | degrees] uncertainty in alignment angle

`apply_beam`(*calc_Q, calc_R, resolution=True*)
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 θ for a fixed wavelength $< \lambda >$, the average of all wavelengths in the probe.

Specifically:

$$
\begin{aligned}
Q_c^2 &= 16\pi(\rho - \rho_{\text{incident}}) \\
Q_i &= Q_c - \delta_i Q_c (i - (n-1)/2) \qquad \text{for } i \in 0 \ldots n-1 \\
\lambda_i &= <\lambda> \\
\theta_i &= \sin^{-1}(Q_i \lambda_i / 4\pi)
\end{aligned}
$$

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.

data

fresnel (*substrate=None*, *surface=None*)
Compute the 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.

name

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 reflectivity

plot_Q4 (***kwargs*)
Plot the Q**4 reflectivity associated with the probe.

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 reflectivity associated with the probe.

plot_linear (***kwargs*)
Plot the data associated with probe.

plot_log(***kwargs*)
Plot the data associated with probe.

plot_residuals(*theory=None*, *suffix=''*, *label=None*, *plot_shift=None*, ***kwargs*)

plot_resolution(*suffix=''*, *label=None*, ***kwargs*)

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 ~ 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=None*, *surface=None*)
Save the data and theory to a file.

scattering_factors(*material*)
Returns the scattering factors associated with the material given the range of wavelengths/energies used in the probe.

simulate_data(*theory*, *noise=2*)
Set the data for the probe to R, adding random noise dR.

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.

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.**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.

refl1d.probe.**spin_asymmetry**(*Qp*, *Rp*, *dRp*, *Qm*, *Rm*, *dRm*)
Compute spin asymmetry for R++, R–.

Parameters:

Qp*, *Rp*, *dRp [vector] Measured ++ cross section and uncertainty.

Qm, Rm, dRm [vector] Measured – cross section and uncertainty.

If *dRp*, *dRm* 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 ΔS_A follows from propagation of error:

$$\Delta S_A^2 = \frac{4(R_{++}^2 \Delta R_{--}^2 + R_{--}^2 \Delta R_{++})}{(R_{++} + R_{--})^4}$$

4.27 refl1d.profile - Model profile

`Microslabs`	Manage the micro slab representation of a model.
`blend`	blend function
`build_mag_profile`	Convert magnetic segments to a smooth profile.
`build_profile`	Convert a step profile to a smooth profile.
`compute_limited_sigma`	

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-σ 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.

4.27.1 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:

```
>>> 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 Pz 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.

```
>>> 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 ...
>>> print "centers = %s ..."%(" ".join("%g"%v for v in Pz[:5]))
centers = 1 3 5 7 9 ...
```

```
>>> 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.

```
>>> 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 ...
>>> 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.

```
>>> print "sigma = %s ..."%(" ".join("%g"%v for v in S.sigma[:5]))
sigma = 0 0 0 0 0 ...
>>> 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 (A)

`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 (A)

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*)

4.28 refl1d.pytwalk - MCMC error analysis using T-Walk steps

pytwalk	This is the t-walk class.

T-walk self adjusting MCMC

class refl1d.pytwalk.**pytwalk**(*n*, *U=<function <lambda> at 0x3cee500>*, *Supp=<function <lambda> at 0x3cee578>*, *ww=[0.0, 0.49180000000000001, 0.49180000000000001, 0.0082000000000000007, 0.0082000000000000007]*, *aw=1.5*, *at=6.0*, *n1phi=4.0*)
This is the t-walk class.

Initiates defining the dimension= n and -log of the objective function= U, Supp defines the support, returns True if x within the support, eg:

Mytwalk = twalk(n=3, U=MyMinusLogf, Supp=MySupportFunction).

Then do: Mytwalk.Run?

Other parameter are: ww= the prob. of choosing each kernel, aw, at, n1phi (see inside twalk.py) with default values as in the paper, normally NOT needed to be changed.

Ana(*par=-1*, *start=0*, *end=0*)
Output Analysis, TS plots, acceptance rates, IAT etc.

GBlowU(*h*, *x*, *xp*)

GHopU(*h*, *x*, *xp*)

Hist(*par=-1*, *start=0*, *end=0*, *g=<function <lambda> at 0x3ceec08>*, *xlab='g'*, *bins=20*)
Basic histograms and output analysis. If par=-1, use g. The function g provides a transformation to be applied to the data, eg g=(lambda x: abs(x[0]-x[1]) would plot a histogram of the distance between parameters 0 and 1, etc.

IAT(*par=-1*, *start=0*, *end=0*, *maxlag=0*)
Calculate the Integrated Autocorrelation Times of parameters par the default value par=-1 is for the IAT of the U's

Run(*T*, *x0*, *xp0*)
Run the twalk.

Run(T, x0, xp0), T = Number of iterations. x0, xp0, two initial points within the support, ***each entry of x0 and xp0 most be different***.

RunRWMH (*T*, *x0*, *sigma*)
Run a simple Random Walk M-H

Save (*fnam*, *start=0*, *thin=1*)
Saves the Output as a text file, starting at start (burn in), with thinning (thin).

SimBlow (*x*, *xp*)

SimHop (*x*, *xp*)

SimTraverse (*x*, *xp*, *beta*)

SimWalk (*x*, *xp*)

Simbeta ()

TS (*par=-1*, *start=0*, *end=0*)
Plot time series of parameter par (default = log f) etc.

onemove (*x*, *u*, *xp*, *up*)
One move of the twalk. This is basically the raw twalk kernel. It is usefull if the twalk is needed inside a more complex MCMC.

onemove(x, u, xp, up), x, xp, two points WITHIN the support ***each entry of x0 and xp0 must be different***. and the value of the objective at x, and xp u=U(x), up=U(xp).

It returns: [y, yp, ke, A, u_prop, up_prop] y, yp: the proposed jump ke: The kernel used, 0=nothing, 1=Walk, 2=Traverse, 3=Blow, 4=Hop A: the M-H ratio u_prop, up_prop: The values for the objective func. at the proposed jumps

4.29 refl1d.quasinewton - BFGS quasi-newton optimizer

quasinewton	Run a quasinewton optimization on the problem.

All modules in this file are implemented from the book “Numerical Methods for Unconstrained Optimization and Nonlinear Equations” by J.E. Dennis and Robert B. Schnabel (Only a few minor modifications are done).

EXAMPLE CALL:

```
n = 2
x0 = [-0.9 0.9]'
fn = lambda p: (1-p[0])**2 + 100*(p[1]-p[0]**2)**2
grad = lambda p: array([-2*(1-p[0]) - 400*(p[1]-p[0]**2)*p[0], 200*p[1]])
Sx = ones(n,1)
typf = 1                       # todo. see what default value is the best
macheps = eps
eta = eps
maxstep = 100
gradtol = 1e-6
steptol = 1e-12                # do not let steptol larger than 1e-9
itnlimit = 1000
result = quasinewton(fn, x0, grad, Sx, typf,
                     macheps, eta, maxstep, gradtol, steptol, itnlimit)
print "status code",result['status']
print "x_min, f(x_min)",result['x'],result['fx']
print "iterations, function calls, linesearch function calls",          result['iterations'],result['
```

`refl1d.quasinewton.`**`quasinewton`**(*fn*, *x0=[ ]*, *grad=[ ]*, *Sx=[ ]*, *typf=1*, *macheps=[ ]*, *eta=[ ]*, *maxstep=100*, *gradtol=9.9999999999999995e-07*, *steptol=9.9999999999999998e-13*, *itnlimit=2000*, *monitor=<function <lambda> at 0x48c0cf8>*)

Run a quasinewton optimization on the problem.

4.30 refl1d.random_lines - random lines and particle swarm optimizers

`random_lines`	

`refl1d.random_lines.`**`random_lines`**(*cfo*, *NP*, *CR=0.90000000000000002*, *epsilon=1e-10*, *maxiter=1000*)

4.31 refl1d.rebin - 1D and 2D rebinning

`rebin`	Rebin a vector.
`rebin2d`	Rebin a matrix.
`test`	

1-D and 2-D rebinning code.

`refl1d.rebin.`**`rebin`**(*x*, *I*, *xo*, *Io=None*, *dtype=<type 'numpy.float64'>*)

Rebin a vector.

x are the existing bin edges xo are the new bin edges I are the existing counts (one fewer than edges)

Io will be used if present, but be sure that it is a contiguous array of the correct shape and size.

dtype is the type to use for the intensity vectors. This can be integer (uint8, uint16, uint32) or real (float32 or f, float64 or d). The edge vectors are all coerced to doubles.

Note that total intensity is not preserved for integer rebinning. The algorithm uses truncation so total intensity will be down on average by half the total number of bins.

`refl1d.rebin.`**`rebin2d`**(*x*, *y*, *I*, *xo*, *yo*, *Io=None*, *dtype=None*)

Rebin a matrix.

x,y are the existing bin edges xo,yo are the new bin edges I is the existing counts (one fewer than edges in each direction)

For example, with x representing the column edges in each row and y representing the row edges in each column, the following represents a uniform field:

```
>>> from refl1d.rebin import rebin2d
>>> x,y = [0,2,4,5], [0,1,3]
>>> z = [[2,2,1],[4,4,2]]
```

We can check this by rebinning with uniform size bins:

```
>>> xo,yo = range(6), range(4)
>>> rebin2d(y,x,z,yo,xo)
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])
```

dtype is the type to use for the intensity vectors. This can be integer (uint8, uint16, uint32) or real (float32 or f, float64 or d). The edge vectors are all coerced to doubles.

Note that total intensity is not preserved for integer rebinning. The algorithm uses truncation so total intensity will be down on average by half the total number of bins.

Io will be used if present, if it is contiguous and if it has the correct shape and type for the input. Otherwise it will raise a TypeError. This will allow you to rebin the slices of an appropriately ordered matrix without making copies.

`refl1d.rebin.`**`test`**`()`

4.32 refl1d.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 Q-dependent resolution function to the theory.
`erf`	Error function calculator.

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

rho_index [integer[M]] *rho* and *irho* columns to use for the various kz.

Returns

***R* | 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}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* | 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=-90.0*, *rho_index=None*`)`

Returns the complex magnetic reflectivity waveform.

See `magnetic_reflectivity` for details.

`refl1d.reflectivity.`**`unpolarized_magnetic`**`(`**args*, ***kw*`)`

Returns the average of magnetic reflectivity for all cross-sections.

See `magnetic_reflectivity` for details.

`refl1d.reflectivity.`**`convolve`**`(`*Qi*, *Ri*, *Q*, *dQ*`)`

Apply Q-dependent resolution function to the theory.

Returns convolution R[k] of width dQ[k] at points Q[k].

`refl1d.reflectivity.`**`erf`**`(`*x*`)`

Error function calculator.

4.33 refl1d.reflmodule - Low level reflectivity calculations

`rebin2d_float32`	rebin2d_float32(xi,yi,Ii,xo,yo,Io): 2-D rebin from (xi,yi) to (xo,yo)
`rebin2d_float64`	rebin2d_float64(xi,yi,Ii,xo,yo,Io): 2-D rebin from (xi,yi) to (xo,yo)
`rebin2d_uint16`	rebin2d_uint16(xi,yi,Ii,xo,yo,Io): 2-D rebin from (xi,yi) to (xo,yo)
`rebin2d_uint32`	rebin2d_uint32(xi,yi,Ii,xo,yo,Io): 2-D rebin from (xi,yi) to (xo,yo)
`rebin2d_uint8`	rebin2d_uint8(xi,yi,Ii,xo,yo,Io): 2-D rebin from (xi,yi) to (xo,yo)
`rebin_float32`	rebin_float32(xi,Ii,xo,Io): rebin from bin edges xi to bin edges xo
`rebin_float64`	rebin_float64(xi,Ii,xo,Io): rebin from bin edges xi to bin edges xo
`rebin_uint16`	rebin_uint16(xi,Ii,xo,Io): rebin from bin edges xi to bin edges xo
`rebin_uint32`	rebin_uint32(xi,Ii,xo,Io): rebin from bin edges xi to bin edges xo
`rebin_uint8`	rebin_uint8(xi,Ii,xo,Io): rebin from bin edges xi to bin edges xo

Reflectometry C Library

`refl1d.reflmodule.`**`rebin2d_float32`**`()`
rebin2d_float32(xi,yi,Ii,xo,yo,Io): 2-D rebin from (xi,yi) to (xo,yo)

`refl1d.reflmodule.`**`rebin2d_float64`**`()`
rebin2d_float64(xi,yi,Ii,xo,yo,Io): 2-D rebin from (xi,yi) to (xo,yo)

`refl1d.reflmodule.`**`rebin2d_uint16`**`()`
rebin2d_uint16(xi,yi,Ii,xo,yo,Io): 2-D rebin from (xi,yi) to (xo,yo)

`refl1d.reflmodule.`**`rebin2d_uint32`**`()`
rebin2d_uint32(xi,yi,Ii,xo,yo,Io): 2-D rebin from (xi,yi) to (xo,yo)

`refl1d.reflmodule.`**`rebin2d_uint8`**`()`
rebin2d_uint8(xi,yi,Ii,xo,yo,Io): 2-D rebin from (xi,yi) to (xo,yo)

`refl1d.reflmodule.`**`rebin_float32`**`()`
rebin_float32(xi,Ii,xo,Io): rebin from bin edges xi to bin edges xo

`refl1d.reflmodule.`**`rebin_float64`**`()`
rebin_float64(xi,Ii,xo,Io): rebin from bin edges xi to bin edges xo

`refl1d.reflmodule.`**`rebin_uint16`**`()`
rebin_uint16(xi,Ii,xo,Io): rebin from bin edges xi to bin edges xo

`refl1d.reflmodule.`**`rebin_uint32`**`()`
rebin_uint32(xi,Ii,xo,Io): rebin from bin edges xi to bin edges xo

`refl1d.reflmodule.`**`rebin_uint8`**`()`
rebin_uint8(xi,Ii,xo,Io): rebin from bin edges xi to bin edges xo

4.34 refl1d.resolution - Resolution

FWHM2sigma	
QL2T	Compute angle from Q and wavelength.
TL2Q	Compute Q from angle and wavelength.
TOF2L	Convert neutron time-of-flight to wavelength.
binedges	Construct bin edges *E* from bin centers *L*.
bins	Return bin centers from low to high preserving a fixed resolution.
binwidths	Determine the wavelength dispersion from bin centers *L*.
dQdL2dT	Convert a calculated Q resolution and wavelength dispersion to
dQdT2dLoL	Convert a calculated Q resolution and angular divergence to a
dTdL2dQ	Convert wavelength dispersion and angular divergence to Q resolution.
divergence	Calculate divergence due to slit and sample geometry.
sigma2FWHM	
slit_widths	Compute the slit widths for the standard scanning reflectometer fixed-opening-fixed geometry.

Resolution calculations

`refl1d.resolution.`**`FWHM2sigma`**`(`*s*`)`

`refl1d.resolution.`**`QL2T`**`(`*Q=None*, *L=None*`)`

Compute angle from Q and wavelength.

$$\theta = \sin^{-1}(|Q|\lambda/4\pi)$$

Returns $\theta°$.

`refl1d.resolution.`**`TL2Q`**`(`*T=None*, *L=None*`)`

Compute Q from angle and wavelength.

$$Q = 4\pi\sin(\theta)/\lambda$$

Returns Q Å^{-1}

`refl1d.resolution.`**`TOF2L`**`(`*d_moderator*, *TOF*`)`

Convert neutron time-of-flight to wavelength.

$$\lambda = (t/d)(h/n_m)$$

where:

λ is wavelength in Å
t is time-of-flight in us
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:

$$L_i = (E_i + E_{i+1})/2 = (E_i + E_i(1+\omega))/2 = E_i(2+\omega)/2$$

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 ω 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 ω 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* Å^{-1} Q and 1-σ Q resolution *L*, *dL* ° angle and FWHM angular divergence

Returns FWHM $\theta, \Delta\theta$

`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 $\Delta\lambda/\lambda$

`refl1d.resolution.`**`dTdL2dQ`**(*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 s_2, 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 [[float] | degrees] Specular measurement angles.

Tlo*, *Thi [float | degrees] Start and end of the opening region. The default if *Tlo* is not specified is to use fixed slits at *slits_below* for all angles.

slits_below*, *slits_above [float OR [float,float] | 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 [float OR [float,float] | mm] Slits at the start of the opening region.

Returns

s1*, *s2 [[float] | mm] Slit widths for each theta.

Slits are assumed to be fixed below angle *Tlo* and above angle *Thi*, 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 *Tlo*, the slits are fixed with widths defined by *slits_below*, which defaults to *slits_at_Tlo*. With no *Thi*, slits are continuously opening above *Tlo*.

Note: This function works equally well if angles are measured in radians and/or slits are measured in inches.

4.35 refl1d.simplex - Nelder-Mead simplex optimizer (amoeba)

`simplex`	Minimize a function using Nelder-Mead downhill simplex algorithm.

Downhill simplex optimizer.

`refl1d.simplex.`**`simplex`**(*f*, *x0=None*, *bounds=None*, *radius=0.050000000000000003*, *xtol=0.0001*, *ftol=0.0001*, *maxiter=None*, *update_handler=None*, *abort_test=<function dont_abort at 0x48d2230>*)

Minimize a function using Nelder-Mead downhill simplex algorithm.

This optimizer is also known as Amoeba (from Numerical Recipes) and the Nealder-Mead simplex algorithm. This is not the simplex algorithm for solving constrained linear systems.

Downhill simplex is a robust derivative free algorithm for finding minima. It proceeds by choosing a set of points (the simplex) forming an n-dimensional triangle, and transforming that triangle so that the worst vertex is improved, either by stretching, shrinking or reflecting it about the center of the triangle. This algorithm is not known for its speed, but for its simplicity and robustness, and is a good algorithm to start your problem with.

Parameters:

f [callable f(x,*args)] The objective function to be minimized.

x0 [ndarray] Initial guess.

bounds [(ndarray,ndarray) or None] Bounds on the parameter values for the function.

radius: float Size of the initial simplex. For bounded parameters (those which have finite lower and upper bounds), radius is clipped to a value in (0,0.5] representing the portion of the range to use as the size of the initial simplex.

Returns: Result (*park.simplex.Result*)

x [ndarray] Parameter that minimizes function.

fx [float] Value of function at minimum: `fopt = func(xopt)`.

iters [int] Number of iterations performed.

calls [int] Number of function calls made.

success [boolean] True if fit completed successfully.

Other Parameters:

xtol [float] Relative error in xopt acceptable for convergence.

ftol [number] Relative error in func(xopt) acceptable for convergence.

maxiter [int=200*N] Maximum number of iterations to perform. Defaults

update_handler [callable] Called after each iteration, as callback(k,n,xk,fxk), where k is the current iteration, n is the maximum iteration, xk is the simplex and fxk is the value of the simplex vertices. xk[0],fxk[0] is the current best.

Notes

Uses a Nelder-Mead simplex algorithm to find the minimum of function of one or more variables.

4.36 refl1d.snsdata - SNS Data

`Liquids`	Loader for reduced data from the SNS Liquids instrument.
`Magnetic`	Loader for reduced data from the SNS Magnetic instrument.
`QRL_to_data`	Convert data to T,L,R
`SNSData`	
`TOF_to_data`	Convert TOF data to neutron probe.
`boltzmann_feather`	Return expected intensity as a function of wavelength given the TOF
`has_columns`	
`intensity_from_spline`	
`load`	Return a probe for SNS data.
`parse_file`	Parse SNS reduced data, returning *header* and *data*.
`write_file`	Save probe as SNS reduced file.

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.

class `refl1d.snsdata.`**`Liquids`**`(`***kw*`)`

Bases: `refl1d.snsdata.SNSData`, `refl1d.instrument.Pulsed`

Loader for reduced data from the SNS Liquids instrument.

`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 *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.

classmethod **`defaults`**`()`

Return default instrument properties as a printable string.

`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.

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

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.

class refl1d.snsdata.**Magnetic**(***kw*)
Bases: refl1d.snsdata.SNSData, refl1d.instrument.Pulsed

Loader for reduced data from the SNS Magnetic instrument.

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 *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.

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

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.

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

`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.

`refl1d.snsdata.`**`QRL_to_data`**(*instrument*, *header*, *data*)
Convert data to T,L,R

class `refl1d.snsdata.`**`SNSData`**
Bases: `object`

`load`(*filename*, ***kw*)

`refl1d.snsdata.`**`TOF_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.

refl1d.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.

refl1d.snsdata.**has_columns**(*header*, *v*)

refl1d.snsdata.**intensity_from_spline**(*Lrange*, *dLoL*, *feather*)

refl1d.snsdata.**load**(*filename*, *instrument=None*, ***kw*)

Return a probe for SNS data.

refl1d.snsdata.**parse_file**(*filename*)

Parse SNS reduced data, returning *header* and *data*.

header dictionary of fields such as 'data', 'title', 'instrument' *data* 2D array of data

refl1d.snsdata.**write_file**(*filename*, *probe*, *original=None*, *date=None*, *title=None*, *notes=None*, *run=None*, *charge=None*)

Save probe as SNS reduced file.

4.37 refl1d.staj - Staj File

MlayerMagnetic	Model definition used by GJ2 program.
MlayerModel	Model definition used by MLayer program.

Read and write staj files

Staj files are the model files for the mlayer and gj2 programs, which are used as the calculation engine for the reflpak suite. Mlayer supports unpolarized beam with multilayer models, and has files ending in **.staj**. GJ2 supports polarized beam without multilayer models, and has files ending in **.sta**.

class refl1d.staj.**MlayerMagnetic**(***kw*)

Bases: object

Model definition used by GJ2 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 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 $\Delta Q/Q = \Delta\lambda/\lambda + \Delta\theta/\theta$

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-σ 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 | $16\pi\rho$, $2\lambda\rho_i$] 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-σ 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 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.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-σ.

`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 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.

classmethod `load`(*filename*)

Load a staj file, returning an MlayerModel object

`save`(*filename*)

Save the staj file

`set`(***kw*)

`sigma_mroughness`

`sigma_roughness`

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:

wavelength*, *wavelength_dispersion*, *angular_divergence resolution is calculated as $\Delta Q/Q = \Delta\lambda/\lambda + \Delta\theta/\theta$

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-σ 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-σ 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:

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.

`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 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.

classmethod `load`(*filename*)

Load a staj file, returning an MlayerModel object

`mu`

`save`(*filename*)

Save the staj file

`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.

4.38 refl1d.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.
mlayer_to_model	Convert a loaded staj file to a refl1d experiment.
model_to_mlayer	Return an mlayer model based on the a slab stack.
save_mlayer	Save a model to a staj file.

Convert staj files to Refl1D models

refl1d.stajconvert.**fit_all**(*M*, *pmp=20*)
Set all non-zero parameters to fitted parameters inside the model.

refl1d.stajconvert.**load_mlayer**(*filename*, *fit_pmp=0*)
Load a staj file as a model.

refl1d.stajconvert.**mlayer_to_model**(*staj*)
Convert a loaded staj file to a refl1d experiment.

Returns a new experiment

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.

refl1d.stajconvert.**save_mlayer**(*experiment*, *filename*, *datafile=None*)
Save a model to a staj file.

4.39 refl1d.stitch - Overlapping reflectivity curve stitching

poisson_average	Compute the poisson average of R/dR using a set of data points.
stitch	Stitch together multiple measurements into one.

Data stitching for reflectometry.

Join together datasets yielding unique sorted Q.

refl1d.stitch.**poisson_average**(*QdQRdRw*)
Compute the poisson average of R/dR using a set of data points.

The returned Q,dQ is the weighted average of the inputs:

```
Q = sum(Q*I)/sum(I)
dQ = sum(dQ*I)/sum(I)
```

The returned R,dR use Poisson averaging:

```
w = sum(y/dy^2)
y = sum((y/dy)^2)/w
dy = sqrt(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 = N/M rate uncertainty: dr = sqrt(N)/M weighted rate: r/dr^2 = (N/M) / (N/M^2) = M weighted rate squared: r^2/dr^2 = (N^2/M^2) / (N/M^2) = N

for two measurements Na, Nb w = ra/dra^2 + rb/drb^2 = Ma + Mb y = ((ra/dra)^2 + (rb/drb)^2)/w = (Na + Nb)/(Ma + Mb) dy = sqrt(y/w) = sqrt((Na + Nb)/ w^2) = sqrt(Na+Nb)/(Ma + Mb)

`refl1d.stitch.`**`stitch`**(*probes*, *same_Q=0.001*, *same_dQ=0.001*)

Stitch together multiple measurements into one.

probes a list of datasets with Q,dQ,R,dR attributes *same_Q* minimum point separation (default is 0.001). *same_dQ* minimum change in resolution that may be averaged (default is 0.001).

Wavelength and angle are not preserved since different points with the same Q,dQ may have different wavelength/angle inputs, particularly for time of flight instruments.

WARNING: the returned Q values may be data dependent, with two measured sets having different Q 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:

```
Q1,dQ1,R1,dR1 = stitch([a1,b1,c1,d1])
Q2,dQ2,R2,dR2 = stitch([a2,b2,c2,d2])
Q2[0],Q2[-1] = Q1[0],Q1[-1] # Force matching end points
R2 = numpy.interp(Q1,Q2,R2)
dR2 = numpy.interp(Q1,Q2,dR2)
Q2 = Q1
```

WARNING: the returned dQ value underestimates the true Q, depending on the relative weights of the averaged data points.

4.40 refl1d.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.

`refl1d.support.`**`get_data_path`**()

Locate the examples directory.

`refl1d.support.`**`sample_data`**(*file*)

4.41 refl1d.util - Miscellaneous functions

`merge_ends`	join the leading and trailing ends of the profile together so fewer
`parse_file`	Parse a file into a header and data.
`indfloat`	Convert string to float, with support for inf and nan.
`auto_shift`	
`next_color`	
`coordinated_colors`	
`dhsv`	Modify color on hsv scale.
`profile`	Profile a function called with the given arguments.
`kbhit`	Check whether a key has been pressed on the console.
`redirect_console`	Console output redirection context
`pushdir`	
`push_seed`	Set the seed value for the random number generator.

`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.

`refl1d.util.`**`parse_file`**(*file*)

Parse a file into a header and data.

Header lines look like # key value Keys can be made multiline by repeating the key Data lines look like float float float Comment lines look like # float float float Data may contain inf or nan values.

Special hack for TOF data: if the first column contains bin edges, then the last row will only have the bin edge. To make the array square, we extend the last row with NaN.

`refl1d.util.`**`indfloat`**(*s*)

Convert string to float, with support for inf and nan.

Example:

```
>>> import numpy
>>> print numpy.isinf(indfloat('inf'))
True
>>> print numpy.isinf(indfloat('-inf'))
True
>>> print numpy.isnan(indfloat('nan'))
True
```

`refl1d.util.`**`auto_shift`**(*offset*)

`refl1d.util.`**`next_color`**()

`refl1d.util.`**`coordinated_colors`**(*base=None*)

`refl1d.util.`**`dhsv`**(*color*, *dh=0*, *ds=0*, *dv=0*, *da=0*)

Modify color on hsv scale.

dv change intensity, e.g., +0.1 to brighten, -0.1 to darken. *dh* change hue *ds* change saturation *da* change transparency

Color can be any valid matplotlib color. The hsv scale is [0,1] in each dimension. Saturation, value and alpha scales are clipped to [0,1] after changing. The hue scale wraps between red to violet.

Example

Make sea green 10% darker:

```
>>> darker = dhsv('seagreen', dv=-0.1)
>>> print [int(v*255) for v in darker]
[37, 113, 71, 255]
```

`refl1d.util.`**`profile`**(*fn*, **args*, ***kw*)
Profile a function called with the given arguments.

`refl1d.util.`**`kbhit`**()
Check whether a key has been pressed on the console.

class `refl1d.util.`**`redirect_console`**(*stdout=None*, *stderr=None*)
Bases: `object`

Console output redirection context

Redirect the console output to a path or file object.

Example

```
>>> print "hello"
hello
>>> with redirect_console("redirect_out.log"):
...     print "hello"
>>> print "hello"
hello
>>> print open("redirect_out.log").read()[:-1]
hello
>>> import os; os.unlink("redirect_out.log")
```

class `refl1d.util.`**`pushdir`**(*path*)
Bases: `object`

class `refl1d.util.`**`push_seed`**(*seed=None*)
Bases: `object`

Set the seed value for the random number generator.

When used in a with statement, the random number generator state is restored after the with statement is complete.

Parameters

seed [int or array_like, optional] Seed for RandomState

Example

Seed can be used directly to set the seed:

```
>>> import numpy
>>> push_seed(24)
<...push_seed object at...>
>>> print numpy.random.randint(0,1000000,3)
[242082    899 211136]
```

Seed can also be used in a with statement, which sets the random number generator state for the enclosed computations and restores it to the previous state on completion:

```
>>> with push_seed(24):
...     print numpy.random.randint(0,1000000,3)
[242082    899 211136]
```

Using nested contexts, we can demonstrate that state is indeed restored after the block completes:

```
>>> with push_seed(24):
...     print numpy.random.randint(0,1000000)
...     with push_seed(24):
...         print numpy.random.randint(0,1000000,3)
...     print numpy.random.randint(0,1000000)
242082
[242082    899 211136]
899
```

The restore step is protected against exceptions in the block:

```
>>> with push_seed(24):
...     print numpy.random.randint(0,1000000)
...     try:
...         with push_seed(24):
...             print numpy.random.randint(0,1000000,3)
...             raise Exception()
...     except:
...         print "Exception raised"
...     print numpy.random.randint(0,1000000)
242082
[242082    899 211136]
Exception raised
899
```

4.42 refl1d.wsolve - Weighted linear and polynomial solver with uncertainty

`LinearModel`	Model evaluator for linear solution to Ax = y.
`PolynomialModel`	Model evaluator for best fit polynomial p(x) = y.
`demo`	
`test`	smoke test...make sure the function continues to return the same
`wpolyfit`	Return the polynomial of degree n that minimizes sum((p(x_i) - y_i)**2/dy_i**2).
`wsolve`	Given a linear system y = A*x + e(dy), estimates x,dx

Solve a potentially over-determined system with uncertainty in the values.

Given: A x = y +/- dy Use: s = wsolve(A,y,dy)

wsolve uses the singular value decomposition for increased accuracy. Estimates the uncertainty for the solution from the scatter in the data.

The returned model object s provides:

> s.x solution s.std uncertainty estimate assuming no correlation s.rnorm residual norm s.DoF degrees of freedom s.cov covariance matrix s.ci(p) confidence intervals at point p s.pi(p) prediction intervals at point p s(p) predicted value at point p

4.42.1 Example

Weighted system:

```
import numpy,wsolve
A = numpy.matrix("1,2,3;2,1,3;1,1,1",'d').A
xin = numpy.array([1,2,3],'d')
dy = numpy.array([0.2,0.01,0.1])
y = numpy.random.normal(numpy.dot(A,xin),dy)
print A,y,dy
s = wsolve.wsolve(A,y,dy)
print "xin,x,dx", xin, s.x, s.std
```

Note there is a counter-intuitive result that scaling the estimated uncertainty in the data does not affect the computed uncertainty in the fit. This is the correct result — if the data were indeed selected from a process with ten times the uncertainty, you would expect the scatter in the data to increase by a factor of ten as well. When this new data set is fitted, it will show a computed uncertainty increased by the same factor. Monte carlo simulations bear this out. The conclusion is that the dataset carries its own information about the variance in the data, and the weight vector serves only to provide relative weighting between the points.

class `refl1d.wsolve.`**`LinearModel`**(*x=None*, *DoF=None*, *SVinv=None*, *rnorm=None*)

Bases: `object`

Model evaluator for linear solution to Ax = y.

Computes a confidence interval (range of likely values for the mean at x) or a prediction interval (range of likely values seen when measuring at x). The prediction interval tells you the width of the distribution at x. This should be the same regardless of the number of measurements you have for the value at x. The confidence interval tells you how well you know the mean at x. It should get smaller as you increase the number of measurements. Error bars in the physical sciences usually show a 1-alpha confidence value of erfc(1/sqrt(2)), representing a 1-sigma standandard deviation of uncertainty in the mean.

Confidence intervals for linear system are given by:

```
x' p +/- sqrt( Finv(1-a,1,df) var(x' p) )
```

where for confidence intervals:

```
var(x' p) = sigma^2 (x' inv(A'A) x)
```

and for prediction intervals:

```
var(x' p) = sigma^2 (1 + x' inv(A'A) x)
```

Stored properties:

```
DoF = len(y)-len(x) = degrees of freedom
rnorm = 2-norm of the residuals y-Ax
x = solution to the equation Ax = y
```

Computed properties:

```
cov = covariance matrix [ inv(A'A); O(n^3) ]
var = parameter variance [ diag(cov); O(n^2)]
std = standard deviation of parameters [ sqrt(var); O(n^2) ]
p = test statistic for chisquare goodness of fit [ chi2.sf; O(1) ]
```

Methods:

```
ci(A,sigma=1):  return confidence interval evaluated at A
pi(A,alpha=0.05):  return prediction interval evaluated at A
```

`ci`(*A*, *sigma=1*)

Compute the calculated values and the confidence intervals for the linear model evaluated at A.

sigma=1 corresponds to a 1-sigma confidence interval

Confidence intervals are sometimes expressed as 1-alpha values, where alpha = erfc(sigma/sqrt(2)).

cov
covariance matrix

p
probability of rejection

pi (*A*, *p=0.050000000000000003*)
Compute the calculated values and the prediction intervals for the linear model evaluated at A.

p = 1-alpha = 0.05 corresponds to 95% prediction interval

std
result standard deviation

var
result variance

class `refl1d.wsolve.`**PolynomialModel** (*s*, *origin=False*)
Bases: `object`

Model evaluator for best fit polynomial p(x) = y.

Stored properties:

```
DoF = len(y)-len(x) = degrees of freedom
rnorm = 2-norm of the residuals y-Ax
coeff = coefficients
degree = polynomial degree
```

Computed properties:

```
cov = covariance matrix [ inv(A'A); O(n^3) ]
var = coefficient variance [ diag(cov); O(n^2)]
std = standard deviation of coefficients [ sqrt(var); O(n^2) ]
p = test statistic for chisquare goodness of fit [ chi2.sf; O(1) ]
```

Methods:

```
__call__(x): return the polynomial evaluated at x
ci(x,sigma=1):  return confidence interval evaluated at x
pi(x,alpha=0.05):  return prediction interval evaluated at x
```

Note that the covariance matrix will not include the ones column if the polynomial goes through the origin.

ci (*x*, *sigma=1*)
Evaluate the polynomial and the confidence intervals at x.

sigma=1 corresponds to a 1-sigma confidence interval

cov
covariance matrix

der (*x*)
Evaluate the polynomial derivative at x.

p
probability of rejection

pi (*x*, *p=0.050000000000000003*)
Evaluate the polynomial and the prediction intervals at x.

p = 1-alpha = 0.05 corresponds to 95% prediction interval

std
result standard deviation

var
result variance

`refl1d.wsolve.`**`demo`**`()`

`refl1d.wsolve.`**`test`**`()`
smoke test...make sure the function continues to return the same result for a particular system.

`refl1d.wsolve.`**`wpolyfit`**`(`*x*, *y*, *dy=1*, *degree=None*, *origin=False*`)`
Return the polynomial of degree n that minimizes sum((p(x_i) - y_i)**2/dy_i**2).

if origin is True, the fit should go through the origin.

`refl1d.wsolve.`**`wsolve`**`(`*A*, *y*, *dy=1*, *rcond=9.9999999999999998e-13*`)`
Given a linear system y = A*x + e(dy), estimates x,dx

A is an n x m array y is an n x k array or vector of length n dy is a scalar or an n x 1 array x is a m x k array

4.43 refl1d.mystic.parameter - Parameters

`Alias`	Parameter alias.
`BaseParameter`	Root of the parameter class, defining arithmetic on parameters
`Constant`	An unmodifiable value.
`Constraint`	Abstract base class for constraints.
`ConstraintEQ`	Constraint operator ==
`ConstraintGE`	Constraint operator >=
`ConstraintGT`	Constraint operator >
`ConstraintLE`	Constraint operator <=
`ConstraintLT`	Constraint operator <
`ConstraintNE`	Constraint operator !=
`Function`	Delayed function evaluator.
`IntegerParameter`	
`OperatorAdd`	Parameter operator +
`OperatorDiv`	Parameter operator /
`OperatorMul`	Parameter operator *
`OperatorPow`	Parameter operator **
`OperatorSub`	Parameter operator -
`Parameter`	A parameter is a symbolic value.
`ParameterSet`	
`Reference`	Create an adaptor so that a model attribute can be treated as if it were a parameter.
`VectorParameter`	
`current`	
`fittable`	Return the list of fittable parameters in no paraticular order.
`flatten`	
`format`	Format parameter set for printing.
`function`	Convert a function into a delayed evaluator.
`randomize`	Set random values to the parameters in the parameter set, with
`substitute`	Return structure a with values substituted for all parameters.
`summarize`	Return a stylized list of parameter names and values with range bars
`unique`	Return the unique set of parameters

Continued on next page

Table 4.1 – continued from previous page

`varying`	Return the list of fitted parameters in the model.

4.43.1 Model parameters

Parameters are a big part of the interface between the model and the fitting engine. By saving and retrieving values and ranges from the parameter, the fitting engine does not need to be aware of the structure of the model.

Users can also perform calculations with parameters, tying together different parts of the model, or different models.

class `refl1d.mystic.parameter.`**`Alias`**(*obj*, *attr*, *p=None*, *name=None*)

Bases: `object`

Parameter alias.

Rather than modifying a model to contain a parameter slot, allow the parameter to exist outside the model. The resulting parameter will have the full parameter semantics, including the ability to replace a fixed value with a parameter expression.

`parameters`()

`update`()

class `refl1d.mystic.parameter.`**`BaseParameter`**

Bases: `object`

Root of the parameter class, defining arithmetic on parameters

`dev`(*sigma=1*, *mu=None*)

Allow the parameter to vary according to a normal distribution, with deviations added to the overall cost function:

dev(sigma, mu) -> Normal(mean=mu,std=sigma)

If mu is None, then it defaults to the current parameter value.

`format`()

Format the parameter, value and range as a string.

`nllf`()

Return the negative log likelihood of seeing the current parameter value.

`parameters`()

`pm`(**args*)

Allow the parameter to vary as value +/- delta.

pm(delta) -> [value-delta, value+delta] pm(plus,minus) -> [value-minus, value+plus]

This uses nice numbers for the resulting range.

`pmp`(**args*)

Allow the parameter to vary as value +/- percent.

pmp(percent) -> [value*(100 - percent)/100, value*(100 + percent)/100] pmp(plus,minus) -> [value*(100 - minus)/100, value*(100 + plus)/100]

This uses nice numbers for the resulting range.

`range`(*low*, *high*)

Allow the parameter to vary within the given range.

`residual`()

Return the negative log likelihood of seeing the current parameter value.

valid()
Return true if the parameter is within the valid range.

class refl1d.mystic.parameter.**Constant**(*value*, *name=None*)
Bases: refl1d.mystic.parameter.BaseParameter

An unmodifiable value.

dev(*sigma=1*, *mu=None*)
Allow the parameter to vary according to a normal distribution, with deviations added to the overall cost function:

dev(sigma, mu) -> Normal(mean=mu,std=sigma)

If mu is None, then it defaults to the current parameter value.

format()
Format the parameter, value and range as a string.

nllf()
Return the negative log likelihood of seeing the current parameter value.

parameters()

pm(**args*)
Allow the parameter to vary as value +/- delta.

pm(delta) -> [value-delta, value+delta] pm(plus,minus) -> [value-minus, value+plus]

This uses nice numbers for the resulting range.

pmp(**args*)
Allow the parameter to vary as value +/- percent.

pmp(percent) -> [value*(100 - percent)/100, value*(100 + percent)/100] pmp(plus,minus) -> [value*(100 - minus)/100, value*(100 + plus)/100]

This uses nice numbers for the resulting range.

range(*low*, *high*)
Allow the parameter to vary within the given range.

residual()
Return the negative log likelihood of seeing the current parameter value.

valid()
Return true if the parameter is within the valid range.

value

class refl1d.mystic.parameter.**Constraint**
Abstract base class for constraints.

class refl1d.mystic.parameter.**ConstraintEQ**(*a*, *b*)
Bases: refl1d.mystic.parameter.Constraint

Constraint operator ==

class refl1d.mystic.parameter.**ConstraintGE**(*a*, *b*)
Bases: refl1d.mystic.parameter.Constraint

Constraint operator >=

class refl1d.mystic.parameter.**ConstraintGT**(*a*, *b*)
Bases: refl1d.mystic.parameter.Constraint

Constraint operator >

class `refl1d.mystic.parameter.`**`ConstraintLE`**(*a*, *b*)
Bases: `refl1d.mystic.parameter.Constraint`

Constraint operator <=

class `refl1d.mystic.parameter.`**`ConstraintLT`**(*a*, *b*)
Bases: `refl1d.mystic.parameter.Constraint`

Constraint operator <

class `refl1d.mystic.parameter.`**`ConstraintNE`**(*a*, *b*)
Bases: `refl1d.mystic.parameter.Constraint`

Constraint operator !=

class `refl1d.mystic.parameter.`**`Function`**(*op*, **args*, ***kw*)
Bases: `refl1d.mystic.parameter.BaseParameter`

Delayed function evaluator.

f.value evaluates the function with the values of the parameter arguments at the time f.value is referenced rather than when the function was invoked.

`args`

`dev`(*sigma=1*, *mu=None*)
Allow the parameter to vary according to a normal distribution, with deviations added to the overall cost function:

dev(sigma, mu) -> Normal(mean=mu,std=sigma)

If mu is None, then it defaults to the current parameter value.

`format`()
Format the parameter, value and range as a string.

`kw`

`nllf`()
Return the negative log likelihood of seeing the current parameter value.

`op`

`parameters`()

`pm`(**args*)
Allow the parameter to vary as value +/- delta.

pm(delta) -> [value-delta, value+delta] pm(plus,minus) -> [value-minus, value+plus]

This uses nice numbers for the resulting range.

`pmp`(**args*)
Allow the parameter to vary as value +/- percent.

pmp(percent) -> [value*(100 - percent)/100, value*(100 + percent)/100] pmp(plus,minus) -> [value*(100 - minus)/100, value*(100 + plus)/100]

This uses nice numbers for the resulting range.

`range`(*low*, *high*)
Allow the parameter to vary within the given range.

`residual`()
Return the negative log likelihood of seeing the current parameter value.

valid()
Return true if the parameter is within the valid range.

value

class refl1d.mystic.parameter.**IntegerParameter**(*value=None*, *bounds=None*, *fixed=None*, *name=None*, ***kw*)
Bases: refl1d.mystic.parameter.Parameter

clip_set(*value*)
Set a new value for the parameter, clipping it to the bounds.

classmethod default(*value*, ***kw*)

dev(*sigma=1*, *mu=None*)
Allow the parameter to vary according to a normal distribution, with deviations added to the overall cost function:

dev(sigma, mu) -> Normal(mean=mu,std=sigma)

If mu is None, then it defaults to the current parameter value.

feasible()
Value is within the limits defined by the model

format()
Format the parameter, value and range as a string.

nllf()
Return the negative log likelihood of seeing the current parameter value.

parameters()

pm(**args*)
Allow the parameter to vary as value +/- delta.

pm(delta) -> [value-delta, value+delta] pm(plus,minus) -> [value-minus, value+plus]

This uses nice numbers for the resulting range.

pmp(**args*)
Allow the parameter to vary as value +/- percent.

pmp(percent) -> [value*(100 - percent)/100, value*(100 + percent)/100] pmp(plus,minus) -> [value*(100 - minus)/100, value*(100 + plus)/100]

This uses nice numbers for the resulting range.

rand(*rng=<module 'numpy.random' from '/usr/lib/python2.6/dist-packages/numpy/random/__init__.pyc'>*)
Set a random value for the parameter.

range(*low*, *high*)
Allow the parameter to vary within the given range.

residual()
Return the negative log likelihood of seeing the current parameter value.

set(*value*)
Set a new value for the parameter, ignoring the bounds.

valid()
Return true if the parameter is within the valid range.

value

class `refl1d.mystic.parameter.`**`OperatorAdd`**(*a*, *b*)

Bases: `refl1d.mystic.parameter.BaseParameter`

Parameter operator +

`dev`(*sigma=1*, *mu=None*)

Allow the parameter to vary according to a normal distribution, with deviations added to the overall cost function:

> dev(sigma, mu) -> Normal(mean=mu,std=sigma)

If mu is None, then it defaults to the current parameter value.

`dvalue`

`format`()

Format the parameter, value and range as a string.

`nllf`()

Return the negative log likelihood of seeing the current parameter value.

`parameters`()

`pm`(**args*)

Allow the parameter to vary as value +/- delta.

pm(delta) -> [value-delta, value+delta] pm(plus,minus) -> [value-minus, value+plus]

This uses nice numbers for the resulting range.

`pmp`(**args*)

Allow the parameter to vary as value +/- percent.

pmp(percent) -> [value*(100 - percent)/100, value*(100 + percent)/100] pmp(plus,minus) -> [value*(100 - minus)/100, value*(100 + plus)/100]

This uses nice numbers for the resulting range.

`range`(*low*, *high*)

Allow the parameter to vary within the given range.

`residual`()

Return the negative log likelihood of seeing the current parameter value.

`valid`()

Return true if the parameter is within the valid range.

`value`

class `refl1d.mystic.parameter.`**`OperatorDiv`**(*a*, *b*)

Bases: `refl1d.mystic.parameter.BaseParameter`

Parameter operator /

`dev`(*sigma=1*, *mu=None*)

Allow the parameter to vary according to a normal distribution, with deviations added to the overall cost function:

> dev(sigma, mu) -> Normal(mean=mu,std=sigma)

If mu is None, then it defaults to the current parameter value.

`dvalue`

`format`()

Format the parameter, value and range as a string.

nllf()
Return the negative log likelihood of seeing the current parameter value.

parameters()

pm(**args*)
Allow the parameter to vary as value +/- delta.

pm(delta) -> [value-delta, value+delta] pm(plus,minus) -> [value-minus, value+plus]

This uses nice numbers for the resulting range.

pmp(**args*)
Allow the parameter to vary as value +/- percent.

pmp(percent) -> [value*(100 - percent)/100, value*(100 + percent)/100] pmp(plus,minus) -> [value*(100 - minus)/100, value*(100 + plus)/100]

This uses nice numbers for the resulting range.

range(*low*, *high*)
Allow the parameter to vary within the given range.

residual()
Return the negative log likelihood of seeing the current parameter value.

valid()
Return true if the parameter is within the valid range.

value

class refl1d.mystic.parameter.**OperatorMul**(*a*, *b*)
Bases: refl1d.mystic.parameter.BaseParameter

Parameter operator *

dev(*sigma=1*, *mu=None*)
Allow the parameter to vary according to a normal distribution, with deviations added to the overall cost function:

> dev(sigma, mu) -> Normal(mean=mu,std=sigma)

If mu is None, then it defaults to the current parameter value.

dvalue

format()
Format the parameter, value and range as a string.

nllf()
Return the negative log likelihood of seeing the current parameter value.

parameters()

pm(**args*)
Allow the parameter to vary as value +/- delta.

pm(delta) -> [value-delta, value+delta] pm(plus,minus) -> [value-minus, value+plus]

This uses nice numbers for the resulting range.

pmp(**args*)
Allow the parameter to vary as value +/- percent.

pmp(percent) -> [value*(100 - percent)/100, value*(100 + percent)/100] pmp(plus,minus) -> [value*(100 - minus)/100, value*(100 + plus)/100]

This uses nice numbers for the resulting range.

range (*low*, *high*)
Allow the parameter to vary within the given range.

residual ()
Return the negative log likelihood of seeing the current parameter value.

valid ()
Return true if the parameter is within the valid range.

value

class refl1d.mystic.parameter.**OperatorPow** (*a*, *b*)
Bases: refl1d.mystic.parameter.BaseParameter

Parameter operator **

dev (*sigma=1*, *mu=None*)
Allow the parameter to vary according to a normal distribution, with deviations added to the overall cost function:

dev(sigma, mu) -> Normal(mean=mu,std=sigma)

If mu is None, then it defaults to the current parameter value.

dvalue

format ()
Format the parameter, value and range as a string.

nllf ()
Return the negative log likelihood of seeing the current parameter value.

parameters ()

pm (**args*)
Allow the parameter to vary as value +/- delta.

pm(delta) -> [value-delta, value+delta] pm(plus,minus) -> [value-minus, value+plus]

This uses nice numbers for the resulting range.

pmp (**args*)
Allow the parameter to vary as value +/- percent.

pmp(percent) -> [value*(100 - percent)/100, value*(100 + percent)/100] pmp(plus,minus) -> [value*(100 - minus)/100, value*(100 + plus)/100]

This uses nice numbers for the resulting range.

range (*low*, *high*)
Allow the parameter to vary within the given range.

residual ()
Return the negative log likelihood of seeing the current parameter value.

valid ()
Return true if the parameter is within the valid range.

value

class refl1d.mystic.parameter.**OperatorSub** (*a*, *b*)
Bases: refl1d.mystic.parameter.BaseParameter

Parameter operator -

dev(*sigma=1*, *mu=None*)
Allow the parameter to vary according to a normal distribution, with deviations added to the overall cost function:

dev(sigma, mu) -> Normal(mean=mu,std=sigma)

If mu is None, then it defaults to the current parameter value.

dvalue

format()
Format the parameter, value and range as a string.

nllf()
Return the negative log likelihood of seeing the current parameter value.

parameters()

pm(**args*)
Allow the parameter to vary as value +/- delta.

pm(delta) -> [value-delta, value+delta] pm(plus,minus) -> [value-minus, value+plus]

This uses nice numbers for the resulting range.

pmp(**args*)
Allow the parameter to vary as value +/- percent.

pmp(percent) -> [value*(100 - percent)/100, value*(100 + percent)/100] pmp(plus,minus) -> [value*(100 - minus)/100, value*(100 + plus)/100]

This uses nice numbers for the resulting range.

range(*low*, *high*)
Allow the parameter to vary within the given range.

residual()
Return the negative log likelihood of seeing the current parameter value.

valid()
Return true if the parameter is within the valid range.

value

class refl1d.mystic.parameter.**Parameter**(*value=None*, *bounds=None*, *fixed=None*, *name=None*, ***kw*)
Bases: refl1d.mystic.parameter.BaseParameter

A parameter is a symbolic value.

It can be fixed or it can vary within bounds.

p = Parameter(3).pmp(10) # 3 +/- 10% p = Parameter(3).pmp(-5,10) # 3 in [2.85,3.3] rounded to 2 digits p = Parameter(3).pm(2) # 3 +/- 2 p = Parameter(3).pm(-1,2) # 3 in [2,5] p = Parameter(3).range(0,5) # 3 in [0,5]

It has hard limits on the possible values, and a range that should live within those hard limits. The value should lie within the range for it to be valid. Some algorithms may drive the value outside the range in order to satisfy soft It has a value which should lie within the range.

Other properties can decorate the parameter, such as tip for tool tip and units for units.

clip_set(*value*)
Set a new value for the parameter, clipping it to the bounds.

classmethod default(*value*, ***kw*)

dev (*sigma=1*, *mu=None*)
Allow the parameter to vary according to a normal distribution, with deviations added to the overall cost function:

dev(sigma, mu) -> Normal(mean=mu,std=sigma)

If mu is None, then it defaults to the current parameter value.

feasible ()
Value is within the limits defined by the model

format ()
Format the parameter, value and range as a string.

nllf ()
Return the negative log likelihood of seeing the current parameter value.

parameters ()

pm (**args*)
Allow the parameter to vary as value +/- delta.

pm(delta) -> [value-delta, value+delta] pm(plus,minus) -> [value-minus, value+plus]

This uses nice numbers for the resulting range.

pmp (**args*)
Allow the parameter to vary as value +/- percent.

pmp(percent) -> [value*(100 - percent)/100, value*(100 + percent)/100] pmp(plus,minus) -> [value*(100 - minus)/100, value*(100 + plus)/100]

This uses nice numbers for the resulting range.

rand (*rng=<module 'numpy.random' from '/usr/lib/python2.6/dist-packages/numpy/random/__init__.pyc'>*)
Set a random value for the parameter.

range (*low*, *high*)
Allow the parameter to vary within the given range.

residual ()
Return the negative log likelihood of seeing the current parameter value.

set (*value*)
Set a new value for the parameter, ignoring the bounds.

valid ()
Return true if the parameter is within the valid range.

class refl1d.mystic.parameter.**ParameterSet** (***kw*)

flatten (*prefix=''*)

class refl1d.mystic.parameter.**Reference** (*obj*, *attr*, *name=None*)
Bases: refl1d.mystic.parameter.Parameter

Create an adaptor so that a model attribute can be treated as if it were a parameter. This allows only direct access, wherein the storage for the parameter value is provided by the underlying model.

Indirect access, wherein the storage is provided by the parameter, cannot be supported since the parameter has no way to detect that the model is asking for the value of the attribute. This means that model attributes cannot be assigned to parameter expressions without some trigger to update the values of the attributes in the model.

clip_set (*value*)
Set a new value for the parameter, clipping it to the bounds.

classmethod default (*value*, ***kw*)

dev (*sigma=1*, *mu=None*)
Allow the parameter to vary according to a normal distribution, with deviations added to the overall cost function:

dev(sigma, mu) -> Normal(mean=mu,std=sigma)

If mu is None, then it defaults to the current parameter value.

feasible ()
Value is within the limits defined by the model

format ()
Format the parameter, value and range as a string.

nllf ()
Return the negative log likelihood of seeing the current parameter value.

parameters ()

pm (**args*)
Allow the parameter to vary as value +/- delta.

pm(delta) -> [value-delta, value+delta] pm(plus,minus) -> [value-minus, value+plus]

This uses nice numbers for the resulting range.

pmp (**args*)
Allow the parameter to vary as value +/- percent.

pmp(percent) -> [value*(100 - percent)/100, value*(100 + percent)/100] pmp(plus,minus) -> [value*(100 - minus)/100, value*(100 + plus)/100]

This uses nice numbers for the resulting range.

rand (*rng=<module 'numpy.random' from '/usr/lib/python2.6/dist-packages/numpy/random/__init__.pyc'>*)
Set a random value for the parameter.

range (*low*, *high*)
Allow the parameter to vary within the given range.

residual ()
Return the negative log likelihood of seeing the current parameter value.

set (*value*)
Set a new value for the parameter, ignoring the bounds.

valid ()
Return true if the parameter is within the valid range.

value

class refl1d.mystic.parameter.**VectorParameter** (*value=None*, *bounds=None*, *fixed=None*, *name=None*, ***kw*)
Bases: refl1d.mystic.parameter.Parameter

clip_set (*value*)
Set a new value for the parameter, clipping it to the bounds.

classmethod default (*value*, ***kw*)

dev(*sigma=1*, *mu=None*)
Allow the parameter to vary according to a normal distribution, with deviations added to the overall cost function:

dev(sigma, mu) -> Normal(mean=mu,std=sigma)

If mu is None, then it defaults to the current parameter value.

feasible()
Value is within the limits defined by the model

format()
Format the parameter, value and range as a string.

nllf()
Return the negative log likelihood of seeing the current parameter value.

parameters()

pm(**args*)
Allow the parameter to vary as value +/- delta.

pm(delta) -> [value-delta, value+delta] pm(plus,minus) -> [value-minus, value+plus]

This uses nice numbers for the resulting range.

pmp(**args*)
Allow the parameter to vary as value +/- percent.

pmp(percent) -> [value*(100 - percent)/100, value*(100 + percent)/100] pmp(plus,minus) -> [value*(100 - minus)/100, value*(100 + plus)/100]

This uses nice numbers for the resulting range.

rand(*rng=<module 'numpy.random' from '/usr/lib/python2.6/dist-packages/numpy/random/__init__.pyc'>*)
Set a random value for the parameter.

range(*low*, *high*)
Allow the parameter to vary within the given range.

residual()
Return the negative log likelihood of seeing the current parameter value.

set(*value*)
Set a new value for the parameter, ignoring the bounds.

valid()
Return true if the parameter is within the valid range.

`refl1d.mystic.parameter.`**current**(*s*)

`refl1d.mystic.parameter.`**fittable**(*s*)
Return the list of fittable parameters in no paraticular order.

Note that some fittable parameters may be fixed during the fit.

`refl1d.mystic.parameter.`**flatten**(*s*)

`refl1d.mystic.parameter.`**format**(*p*, *indent=0*)
Format parameter set for printing.

Note that this only says how the parameters are arranged, not how they relate to each other.

`refl1d.mystic.parameter.`**`function`**(*op*)

Convert a function into a delayed evaluator.

The value of the function is computed from the values of the parameters at the time that the function value is requested rather than when the function is created.

`refl1d.mystic.parameter.`**`randomize`**(*s*)

Set random values to the parameters in the parameter set, with values chosen according to the bounds.

`refl1d.mystic.parameter.`**`substitute`**(*a*)

Return structure a with values substituted for all parameters.

The function traverses lists, tuples and dicts recursively. Things which are not parameters are returned directly.

`refl1d.mystic.parameter.`**`summarize`**(*pars*)

Return a stylized list of parameter names and values with range bars suitable for printing.

`refl1d.mystic.parameter.`**`unique`**(*s*)

Return the unique set of parameters

The ordering is stable. The same parameters/dependencies will always return the same ordering, with the first occurrence first.

`refl1d.mystic.parameter.`**`varying`**(*s*)

Return the list of fitted parameters in the model.

This is the set of parameters that will vary during the fit.

4.44 refl1d.mystic.bounds - Bounds

`pm`	Return the tuple (~v-dv,~v+dv), where ~expr is a 'nice' number near to to the value of expr.
`pmp`	Return the tuple (~v-%v,~v+%v), where ~expr is a 'nice' number near to the value of expr.
`pm_raw`	Return the tuple [v-dv,v+dv].
`pmp_raw`	Return the tuple [v-%v,v+%v]
`nice_range`	Given a range, return an enclosing range accurate to two digits.
`init_bounds`	Returns a bounds object of the appropriate type given the arguments.
`Unbounded`	Unbounded parameter.
`Bounded`	Bounded range.
`BoundedAbove`	Semidefinite range bounded above.
`BoundedBelow`	Semidefinite range bounded below.
`Distribution`	Parameter is pulled from a distribution.
`Normal`	Parameter is pulled from a normal distribution.
`SoftBounded`	Parameter is pulled from a stretched normal distribution.

Parameter bounds.

Parameter bounds encompass several features of our optimizers.

First and most trivially they allow for bounded constraints on parameter values.

Secondly, for parameter values known to follow some distribution, the bounds encodes a penalty function as the value strays from its nominal value. Using a negative log likelihood cost function on the fit, then this value naturally contributes to the overall likelihood measure.

Predefined bounds are:

```
Unbounded
    range (-inf, inf)
BoundedBelow
    range (base, inf)
```

```
BoundedAbove
    range (-inf, base)
Bounded
    range (low, high)
SoftBounded
    range (low, high) with gaussian probability sigma
Normal
    range (-inf, inf) with gaussian probability
```

Using `int_bounds()` you can create the appropriate bounded or unbounded object for the kind of input.

New bounds can be defined following the abstract base class interface defined in `Bounds`.

For generating bounds given a value, we provide a few helper functions:

```
v +/- d:  pm(x,dx) or pm(x,-dm,+dp) or pm(x,+dp,-dm)
    return (x-dm,x+dm) limited to 2 significant digits
v +/- p%: pmp(x,p) or pmp(x,-pm,+pp) or pmp(x,+pp,-pm)
    return (x-pm*x/100, x+pp*x/100) limited to 2 sig. digits
pm_raw(x,dx) or raw_pm(x,-dm,+dp) or raw_pm(x,+dp,-dm)
    return (x-dm,x+dm)
pmp_raw(x,p) or raw_pmp(x,-pm,+pp) or raw_pmp(x,+pp,-pm)
    return (x-pm*x/100, x+pp*x/100)
nice_range(lo,hi)
    return (lo,hi) limited to 2 significant digits
```

`refl1d.mystic.bounds.`**`pm`**`(`*v*, **args*`)`

Return the tuple (~v-dv,~v+dv), where ~expr is a 'nice' number near to to the value of expr. For example:

```
>>> r = pm(0.78421, 0.0023145)
>>> print "%g - %g"%r
0.7818 - 0.7866
```

If called as pm(value, +dp, -dm) or pm(value, -dm, +dp), return (~v-dm, ~v+dp).

`refl1d.mystic.bounds.`**`pmp`**`(`*v*, **args*`)`

Return the tuple (~v-%v,~v+%v), where ~expr is a 'nice' number near to the value of expr. For example:

```
>>> r = pmp(0.78421, 10)
>>> print "%g - %g"%r
0.7 - 0.87
>>> r = pmp(0.78421, 0.1)
>>> print "%g - %g"%r
0.7834 - 0.785
```

If called as pmp(value, +pp, -pm) or pmp(value, -pm, +pp), return (~v-pm%v, ~v+pp%v).

`refl1d.mystic.bounds.`**`pm_raw`**`(`*v*, **args*`)`

Return the tuple [v-dv,v+dv].

If called as pm_raw(value, +dp, -dm) or pm_raw(value, -dm, +dp), return (v-dm, v+dp).

`refl1d.mystic.bounds.`**`pmp_raw`**`(`*v*, **args*`)`

Return the tuple [v-%v,v+%v]

If called as pmp_raw(value, +pp, -pm) or pmp_raw(value, -pm, +pp), return (v-pm%v, v+pp%v).

`refl1d.mystic.bounds.`**`nice_range`**`(`*range*`)`

Given a range, return an enclosing range accurate to two digits.

`refl1d.mystic.bounds.`**`init_bounds`**`(`*v*`)`

Returns a bounds object of the appropriate type given the arguments.

This is a helper factory to simplify the user interface to parameter objects.

class `refl1d.mystic.bounds.`**`Unbounded`**

Bases: `refl1d.mystic.bounds.Bounds`

Unbounded parameter.

The random initial condition is assumed to be between 0 and 1

The probability is uniformly 1/inf everywhere, which means the negative log likelihood of P is inf everywhere. A value inf will interfere with optimization routines, and so we instead choose P == 1 everywhere.

Convert sign*m*2^e to sign*(e+1023+m), yielding a value in [-2048,2048]. This can then be converted to a value in [0,1].

`get01`(*x*)

`getfull`(*x*)

`nllf`(*value*)

`put01`(*v*)

`putfull`(*v*)

`random`(*n=1*)

`residual`(*value*)

`start_value`()

Return a default starting value if none given.

class `refl1d.mystic.bounds.`**`Bounded`**(*lo*, *hi*)

Bases: `refl1d.mystic.bounds.Bounds`

Bounded range.

[lo,hi] <-> [0,1] scale is simple linear [lo,hi] <-> (-inf,inf) scale uses exponential expansion

While technically the probability of seeing any value within the range is 1/range, for consistency with the semi-infinite ranges and for a more natural mapping between nllf and chisq, we instead set the probability to 0. This choice will not affect the fits.

`get01`(*x*)

`getfull`(*x*)

`nllf`(*value*)

`put01`(*v*)

`putfull`(*v*)

`random`(*n=1*)

`residual`(*value*)

`start_value`()

Return a default starting value if none given.

class `refl1d.mystic.bounds.`**`BoundedAbove`**(*base*)

Bases: `refl1d.mystic.bounds.Bounds`

Semidefinite range bounded above.

[-inf,base] <-> [0,1] uses logarithmic compression [-inf,base] <-> (-inf,inf) is direct below base-1, 1/(base-x) above

Logarithmic compression works by converting sign*m*2^e+base to sign*(e+1023+m), yielding a value in [0,2048]. This can then be converted to a value in [0,1].

Note that the likelihood function is problematic: the true probability of seeing any particular value in the range is infintesimal, and that is indistinguishable from values outside the range. Instead we say that P = 1 in range, and 0 outside.

get01(*x*)

getfull(*x*)

nllf(*value*)

put01(*v*)

putfull(*v*)

random(*n=1*)

residual(*value*)

start_value()

class `refl1d.mystic.bounds.`**BoundedBelow**(*base*)
Bases: `refl1d.mystic.bounds.Bounds`

Semidefinite range bounded below.

The random initial condition is assumed to be within 1 of the maximum.

[base,inf] <-> (-inf,inf) is direct above base+1, -1/(x-base) below [base,inf] <-> [0,1] uses logarithmic compression.

Logarithmic compression works by converting sign*m*2^e+base to sign*(e+1023+m), yielding a value in [0,2048]. This can then be converted to a value in [0,1].

Note that the likelihood function is problematic: the true probability of seeing any particular value in the range is infintesimal, and that is indistinguishable from values outside the range. Instead we say that P = 1 in range, and 0 outside.

get01(*x*)

getfull(*x*)

nllf(*value*)

put01(*v*)

putfull(*v*)

random(*n=1*)

residual(*value*)

start_value()

class `refl1d.mystic.bounds.`**Distribution**(*dist*)
Bases: `refl1d.mystic.bounds.Bounds`

Parameter is pulled from a distribution.

dist must implement the distribution interface from scipy.stats. In particular, it should define methods rvs, nnlf, cdf and ppf and attributes args and dist.name.

get01(*x*)

getfull(*x*)

nllf(*value*)

put01(*v*)

putfull(*v*)

random(*n=1*)

residual(*value*)

start_value()
Return a default starting value if none given.

class refl1d.mystic.bounds.**Normal**(*mean=0*, *std=1*)
Bases: refl1d.mystic.bounds.Distribution

Parameter is pulled from a normal distribution.

If you have measured a parameter value with some uncertainty (e.g., the film thickness is 35+/-5 according to TEM), then you can use this measurement to restrict the values given to the search, and to penalize choices of this fitting parameter which are different from this value.

mean is the expected value of the parameter and *std* is the 1-sigma standard deviation.

get01(*x*)

getfull(*x*)

nllf(*value*)

put01(*v*)

putfull(*v*)

random(*n=1*)

residual(*value*)

start_value()
Return a default starting value if none given.

class refl1d.mystic.bounds.**SoftBounded**(*lo*, *hi*, *std=1*)
Bases: refl1d.mystic.bounds.Bounds

Parameter is pulled from a stretched normal distribution.

This is like a rectangular distribution, but with gaussian tails.

The intent of this distribution is for soft constraints on the values. As such, the random generator will return values like the rectangular distribution, but the likelihood will return finite values based on the distance from the from the bounds rather than returning infinity.

Note that for bounds constrained optimizers which force the value into the range [0,1] for each parameter we don't need to use soft constraints, and this acts just like the rectangular distribution.

get01(*x*)

getfull(*x*)

nllf(*value*)

put01(*v*)

putfull(*v*)

random(*n=1*)

residual(*value*)

start_value()
Return a default starting value if none given.

4.45 refl1d.mystic.formatnum - Format numbers

format_uncertainty	Value and uncertainty formatter.
format_uncertainty_pm	Given *value* v and *uncertainty* dv, return a string v +/- dv.
format_uncertainty_compact	Given *value* v and *uncertainty* dv, return the compact

4.45.1 Number formatting

Format values and uncertainties nicely for printing.

format_uncertainty_pm() produces the expanded format v +/- err.

format_uncertainty_compact() produces the compact format v(##), where the number in parenthesis is the uncertainty in the last two digits of v.

format_uncertainty() uses the compact format by default, but this can be changed to use the expanded +/- format by setting format_uncertainty.compact to False.

The formatted string uses only the number of digits warranted by the uncertainty in the measurement.

If the uncertainty is 0 or not otherwise provided, the simple %g floating point format option is used.

Infinite and indefinite numbers are represented as inf and NaN.

Example:

```
>>> v,dv = 757.2356,0.01032
>>> print format_uncertainty_pm(v,dv)
757.236 +/- 0.010
>>> print format_uncertainty_compact(v,dv)
757.236(10)
>>> print format_uncertainty(v,dv)
757.236(10)
>>> format_uncertainty.compact = False
>>> print format_uncertainty(v,dv)
757.236 +/- 0.010
```

UncertaintyFormatter() returns a private formatter with its own formatter.compact flag.

refl1d.mystic.formatnum.**format_uncertainty_pm**(*value*, *uncertainty*)
Given *value* v and *uncertainty* dv, return a string v +/- dv.

refl1d.mystic.formatnum.**format_uncertainty_compact**(*value*, *uncertainty*)
Given *value* v and *uncertainty* dv, return the compact representation v(##), where ## are the first two digits of the uncertainty.

Modules defined within Refl1D

refl1d.abeles	Optical matrix form of the reflectivity calculation.
refl1d.bspline	BSpline calculator.
refl1d.cheby	Freeform modeling with Chebyshev polynomials
refl1d.cli	
refl1d.dist	Inhomogeneous samples
refl1d.errors	Visual representation of model uncertainty.

Continued on next page

Table 4.2 – continued from previous page

refl1d.experiment	Experiment definition
refl1d.fitproblem	Interface between the models and the fitters.
refl1d.fitservice	Fit job definition for the distributed job queue.
refl1d.fitters	
refl1d.freeform	Freeform modeling with B-Splines
refl1d.fresnel	Pure python Fresnel reflectivity calculator.
refl1d.garefl	Load garefl models into refl1d
refl1d.initpop	Population initialization routines.
refl1d.instrument	Reflectometry instrument definitions.
refl1d.magnetic	Magnetic modeling for 1-D reflectometry.
refl1d.mapper	
refl1d.material	Reflectometry materials.
refl1d.materialdb	Common materials in reflectometry experiments along with densities.
refl1d.model	Reflectometry models
refl1d.mono	Monotonic spline modeling for free interfaces
refl1d.ncnrdata	NCNR data loaders
refl1d.numpyerrors	Decorator for handling numpy errors.
refl1d.partemp	Parallel tempering for continuous function optimization and uncertainty analysis
refl1d.polymer	Layer models for polymer systems.
refl1d.probe	Experimental probe.
refl1d.profile	Scattering length density profile.
refl1d.pytwalk	T-walk self adjusting MCMC
refl1d.quasinewton	All modules in this file are implemented from the book
refl1d.random_lines	
refl1d.rebin	1-D and 2-D rebinning code.
refl1d.reflectivity	Basic reflectometry calculations
refl1d.reflmodule	Reflectometry C Library
refl1d.resolution	Resolution calculations
refl1d.simplex	Downhill simplex optimizer.
refl1d.snsdata	SNS data loaders
refl1d.staj	Read and write staj files
refl1d.stajconvert	Convert staj files to Refl1D models
refl1d.stitch	Data stitching for reflectometry.
refl1d.support	Support files for the application.
refl1d.util	
refl1d.wsolve	Solve a potentially over-determined system with uncertainty in the values.
refl1d.mystic.parameter	Model parameters
refl1d.mystic.bounds	Parameter bounds.
refl1d.mystic.formatnum	Number formatting

PYTHON MODULE INDEX

INDEX